Hardware-Accelerated Super-Resolution

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Electrical and Computer Engineering,
Computer Science

WPI

By: Will Buchta, Diyar Aljabbari,
Parker Langa, Parker Lavering, Adam Spencer

Project Advisor: Professor Patrick Schaumont, PhD.

Date: May 2025

Abstract

This project brings machine learning and hardware development together to implement real-time video upscaling algorithms on FPGAs. Video frames from a Nintendo Wii are analyzed for variance and routed to either bilinear interpolation or a convolutional neural network for upscaling. Hardware was developed and verified in Vitis HLS C++. The two development boards used for testing and implementing the design were the KV260 and ZCU102. This report outlines the team's successes and identifies areas for improvement in future designs.

Executive Summary

Introduction

Despite the rapid advancement of display technology and the release of monitors with higher resolutions almost every year, old media, such as retro video games, remain at the low resolution at which they were created. These games were designed for low-resolution displays available at the time of their creation and do not scale well to new screens on the market. One way to accommodate newer displays is to upscale the video with super-resolution algorithms, creating a new image that is larger and higher quality than the original. The capabilities of software alone are insufficient to achieve this goal in real-time, and hardware lacks the ease of use and flexibility that software offers. This project addresses the challenge of upscaling low-resolution video in real-time by developing a hardware/software coprocessor pipeline for FPGAs that consists of variance measurement, color space conversion, CNN upscaling, and bilinear interpolation.

The goals of this project are as follows.

1. Achieve real-time 2x video upscaling on the output of the Nintendo Wii, targeting 25 FPS and a resolution of 1440 x 1152 at the output
2. Train a super-resolution model to upscale native Wii data by a factor of two
3. Implement a hybrid upscaling system, utilizing FSRCNN [1] and bilinear interpolation based on the variance of image tiles
4. Design and verify hardware IP using Vitis HLS
5. Optimize the hardware implementation to fit the resource constraints of the ZCU102 development board

This work was completed over the course of one academic year, during which the team designed, implemented, verified, and optimized a super-resolution pipeline on an FPGA. This project was student-led with the help of Professor Patrick Schaumont.

Design and Methodology

Since FPGAs have limited on-chip resources, the main idea is to split an image into subsections, or "tiles," and process them based on their level of detail. If a tile has a high variance, it contains many different colors and a lot of detail, and should be upscaled via a slower, high-quality algorithm. If a tile has little variation, for example, an image of a blue sky, it should be upscaled as quickly as possible using a lower-quality but faster method. This division enables the high-quality method to be computationally expensive while avoiding wasting clock cycles on unnecessary tiles. These two methods are FSRCNN [1], a convolutional neural network designed for super-resolution, and bilinear interpolation, a simpler image upscaling technique. This paper details the design process and implementation of three main IP blocks: one designed to measure tile variance and convert color space, another to run FSRCNN, and a third to perform bilinear interpolation.

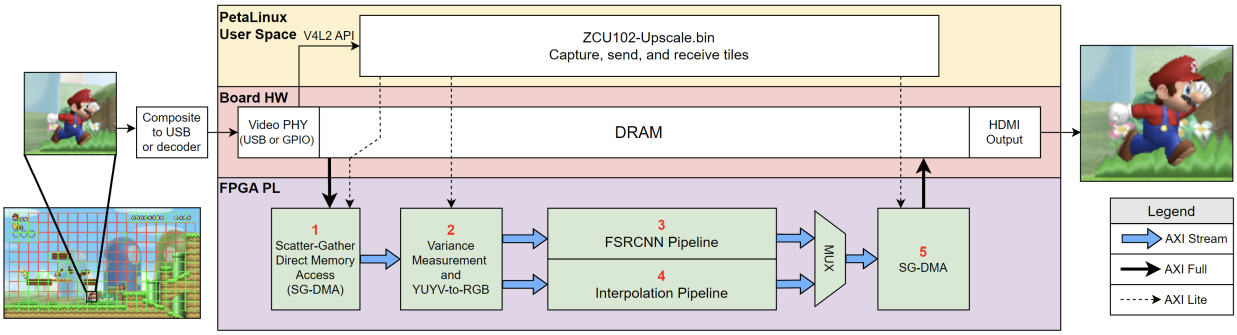

Figure 1: A system-level diagram of the upscaling process.

Additionally, it outlines the build infrastructure, software, and machine learning development surrounding these components. Figure 1 details the high-level data flow of this project.

Results

The team currently has a demonstration able to run bilinear interpolation on real Wii frames at 9.7 FPS with the KV260 development board. The physical interface for video input on the ZCU102 is still under development, so that demo currently only upscales a single image from a file. As for the convolution IP, it uses more resources than what is available on the KV260, so it will only synthesize for the ZCU102. The IP is functionally verified in both Python and Vitis HLS; however, the team is currently debugging issues that arise when running cosimulation and testing on hardware.

A significant bottleneck in our system was not fully identified until late in the project: transferring data to and from the programmable logic. This is because the DMA engine needs to fetch buffer descriptors every four stream beats. The latency to send a single 32 x 32 pixel tile into the FPGA is roughly 1,700 clock cycles, or approximately 17 µs at 100 MHz. Receiving a tile takes approximately 3,600 clock cycles, or 36 µs at 100 MHz. With 368 tiles, the total latency because of data transfer is 19,500 µs, or nearly 20 ms. At 25 FPS, this is half of the total budget to achieve real-time processing. Additionally, it takes 54 clock cycles to transfer one row of a tile, with only four of those clock cycles being used to actually send the data. This means that 93% of the transfer time is spent fetching buffer descriptors from DRAM.

Conclusions and Recommendations

The outcome of the team's work on this project proves that real-time video upscaling on an FPGA is attainable. While only one of the upscaling methods was successfully implemented on hardware, the entire design is fully verified in software. Moving forward with the project, optimizations regarding efficiency, both in terms of speed and resource utilization, as well as the quality of results, should be applied to both upscaling algorithms. This project was successful in bringing multiple areas of computer science and computer engineering together to create an embedded system.

Acknowledgments

The team would like to thank the following for help throughout the project.

Professor Patrick Schaumont for guidance

John Eismeier for helping with IT infrastructure

Brian Jackson for providing suggestions for model development

The AMD University Partnership Program for donating a ZCU102 evaluation board

Statement of Authorship

Section	**Subsection**	**Author(s)**
Literature Review	Image Upscaling	Langa
	Video Signal Properties	Spencer
	Variance	Spencer
	Convolutional Neural Network	Langa
	Hardware Implementation of Upscaling Techniques	Langa
	Verification and Universal Verification Methodology	Aljabbari
Design	Functional Architecture	Buchta, Lavering
	Generating Weights	Lavering
	Managing Dataflow in Hardware	Buchta
	HLS: Variance and Color Space Conversion	Spencer
	HLS: Bilinear Interpolation	Langa
	HLS: FSRCNN	Buchta
	Software Development	Buchta
	Project Infrastructure	Buchta
	HLS Verification Methods	Aljabbari
	Software Development	Buchta
Results	System Overview	Buchta
	Verification Results	Aljabbari
	Example Upscaled Image	Lavering, Buchta
Other	Introduction	Langa, Spencer
	Discussion	Spencer, Lavering
	Future Work	Buchta, Spencer, Lavering, Langa
	Conclusion	Langa

Contents

List of Tables

List of Figures

1 Introduction

Developments in computer monitors over the past 20 years have led to an increased adoption of higher-resolution displays by the general public [10]. Television production has shown a similar increase with the adoption of 4K TVs in residential households in the US increasing from 1% in 2014 to 31% in 2018 [11]. Despite this, much of the available media does not utilize the full resolution available on these displays. Graphics industry leaders, such as Nvidia, are capitalizing on this trend by developing super-resolution algorithms compatible with their GPUs, which can enhance the quality of low-resolution videos provided by streaming services through AI upscaling [12]. However, these advancements generally focus on media that is already in Full High Definition, which is 1920 x 1080 pixels. Simultaneously, there is a continued interest in retro video games such as *Super Mario Bros., Minesweeper, Tetris, and Final Fantasy* [13]. These factors combine to illustrate a demand in the market for creating high-level video resolution methods for low-resolution source material that function in real-time.

Retro video games present not only an upscaling challenge due to their low resolution, but also because of the distinctive art style employed. Machine learning applications of AI super-resolution have primarily focused on real-world images, as seen in previous research papers [1][14][15]. However, video game inputs differ in style and complexity. While the original art style should be maintained, the upscaling method must enhance the image's detail, rather than simply increasing the number of pixels. Additionally, latency needs to be minimized to provide a seamless and enjoyable user experience. Upscaling applications using a combination of hardware and software has been the focus of several research projects; however, these efforts have yet to specifically target video game applications [1][14][15].

The objective of this project was to replicate the work laid out in "FPGA-Based Real-Time Super-Resolution System for Ultra High Definition Videos" [15] and implement a hardware accelerator for super-resolution targeted at the New Super Mario Bros. Wii video game under the following constraints:

1. Ensure the architecture is fast enough to create a real-time output
2. Create a design whose resource utilization fits on the selected hardware
3. Complete the project within an eight-month time frame

The native output of the Nintendo Wii is 576i, spanning 720 pixels horizontally and 576 pixels vertically. Given the level of upscaling necessary to increase a standard-definition input of 576i by a factor of two to 1440 x 1152, software alone cannot operate with sufficient efficiency to perform the required computations within the constraints of a real-time application. The project was first implemented on a Kria KV260 Vision AI and later on a ZCU102 Zynq UltraScale+™ MPSoC development board.

Figure 1 details the high-level flow that constitutes our system. Due to hardware limitations, input images are upscaled as a series of smaller 32 x 32 pixel tiles. The system has five main steps: programming a DMA engine to send an image frame down as a series of tiles, performing a variance measurement on each tile, converting it from the source (YUV) to destination format (RGB), upscaling with bilinear interpolation or the Fast Super-Resolution Convolutional Neural Network (FSRCNN), and using scatter-gather DMA to transfer the upscaled tiles back to a frame buffer in DRAM [5][7]. If an image tile has a high variance, it is sent through the computationally expensive FSRCNN pipeline; else, it is interpolated. Each major IP block was developed using Vitis High-Level Synthesis (HLS) to allow for rapid design iterations. The creation of IP blocks also required verifying their functionality, for which the team created a series of testbenches.

This report outlines the steps taken by the team to address the complex process of hardware-accelerated image upscaling. A list of acronyms (Appendix A) is provided to aid the reader in understanding various image processing terminology. Section 2 discusses the prior research necessary to understand the image processing field and the insights it yields for the team's implementation. Section 3 outlines the design process and methods for each subset of the project, specifically focusing on aspects such as machine learning, variance, color space conversion, bilinear interpolation, FSRCNN, application software development, and build infrastructure. Section 4 describes the verification methodology for each IP unit. After individual IP outcomes are described in their relevant design section, the results section details the system-wide outcomes.

The report concludes with a discussion about ethical concerns, future work, and a summary of all project work. This project was student-led with the help of Professor Patrick Schaumont.

2 Literature Review

The literature review for this report details the background knowledge necessary to successfully implement the project goals. It begins by providing a broad overview of image upscaling, then delves deeper to offer information about video signal processing. Next, it discusses the theoretical and mathematical background for variance calculations, interpolation upscaling schemes, and convolutional neural networks. It then details hardware design concepts such as the systems available, as well as parallelization and pipelining. The hardware implementation also provides insight into HLS and its applications in FPGA development. Finally, it discusses verification strategies for projects that involve both hardware and software components.

2.1 Image Upscaling

Image upscaling is the process of taking images or videos and increasing their resolution to increase size and/or improve quality. While this may seem straightforward, to perform this increase, new information must be added to the image by adding new pixels. The process of increasing an image beyond its original dimensions or resolution is referred to as super-resolution, and is the focus of significant research due to its complexity [16]. Additionally, the issue of upscaling images has a quadratic complexity, given that for every scale factor of two, the number of pixels increases by a factor of four. The core of modern super-resolution techniques lies at the intersection of hardware and software, with hardware development focusing on efficient implementation and software development focusing on leveraging machine learning to enhance the value of upscaled images beyond their original content. Both hardware and software developers require an understanding of image processing and prior work to successfully implement an upscaling algorithm.

Those approaching the problem from a hardware perspective must understand the complexities and potential bottlenecks of upscaling approaches, as well as the intensive calculations that some methods require. Additionally, identifying bottlenecks allows for the allocation of resources to overcome them and create speed-ups.

The work previously done in image processing, regardless of whether it focused on super-resolution, allows an informed approach to system design and implementation. There is currently a wide variety of applications utilizing image processing, ranging from using image filters in video games as shaders to object detection and facial recognition. Applications that utilize detection or recognition focus on manipulating or interpreting pixel values by adding external information, and although they do not increase the size of the image, they resemble convolutional neural network (CNN) methods used for image upscaling [17][18]. As such, research surrounding convolutional neural networks for image processing was completed before the development of a methodology.

2.2 Video Signal Properties

The two main components of a video are its **frame rate** and **resolution**. Frame rate refers to the number of images displayed per second, creating the illusion of motion; a higher frame rate results in a smoother video. Resolution indicates how many pixels span the horizontal and vertical axes of each frame, and a higher resolution means a higher perceived quality of the video being displayed. By testing and measuring the frame rate and resolution output from the Wii in software, the team found that it outputs 25 frames per second (FPS) at a resolution of 576i. These two metrics helped align the project's goals, as they quantified the required computation data rate for real-time processing, as well as the initial and final desired dimensions of the frames to be upscaled. By definition, real-time video upscaling involves manipulating video under two simple conditions: retaining the original timing and increasing the video's scale. The success of this project is largely quantified by these two parameters once all elements are implemented. Table 1 enumerates the input resolution and frame rate, as well as the desired output resolution and frame rate of the super-resolution system.

Color space describes a standardized manner of representing colors for image or video formats. Images and video are split up into pixels, which can be represented in various color spaces. Each color space

Table 1: Input and output desired resolution and frame rate for the super-resolution system.

Device	Horizontal Pixel Count	Vertical Pixel Count	Frame Rate	Pixels/second
Nintendo Wii	720	576	25	10,368,000
Desired Super-Resolution Output	1440	1152	25	41,472,000

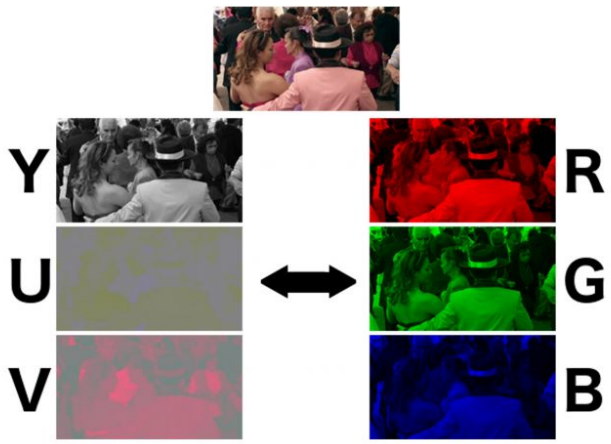

Figure 2: YUV and RGB color space channel visualization [2].

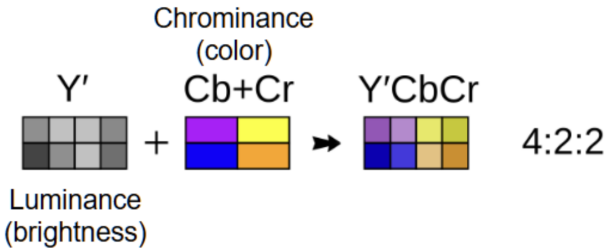

Figure 3: YUV 4:2:2 channel breakdown [3].

has a way of dividing pixel data into separate components, which combine to create a color that can be rendered on a screen. An image can have hundreds of thousands or even millions of pixels, each capable of representing millions of different colors.

Two of the most common color spaces used in video applications are YUV and RGB. YUV consists of three channels: luminance, chrominance-blue, and chrominance-red. The Y channel, or luminance, of a YUV pixel represents its brightness level, resulting in a grayscale mapping of the image when displayed independently. The U and V channels express the amount of blue and red that belongs in each pixel, respectively. RGB uses three channels as well, being the amount of red, green, and blue in each pixel. In any color space, each of these channels is primarily represented by an 8-bit number, meaning that the values range from 0 to 255. Figure 2 illustrates the difference between the YUV and RGB color spaces, showing what each channel looks like when displayed individually.

Both of these color spaces have many different variations, using different bit widths per channel, or shared channels between pixels, but the variations that are utilized in this design are YUV 4:2:2, RGB 8:8:8, and RGB 5:6:5. As seen in Figure 3, the 4:2:2 variation of YUV uses chrominance sampling on every other pixel, reducing the amount of memory each pixel takes up, while subsequently reducing quality [19]. YUV 4:2:2 requires 32 bits of memory for every two pixels, making it more memory-efficient compared to the RGB 8:8:8 standard. The only difference between RGB 8:8:8 and RGB 5:6:5 is the number of bits used per channel; RGB 8:8:8 uses 8 bits per channel, while RGB 5:6:5 uses 5 bits for red, 6 bits for green, and 5 bits for blue, making each pixel use a total of 16 bits instead of 24.

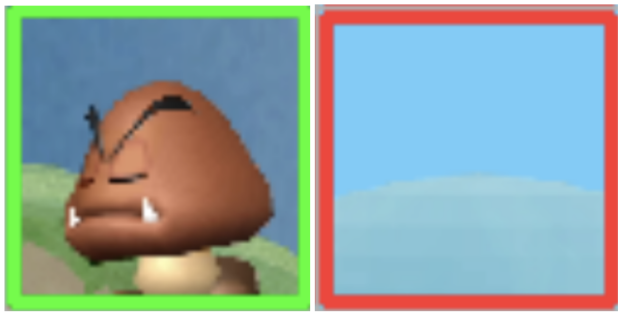

Figure 4: An example of a high variance tile (left) and a low variance tile (right) taken from *New Super Mario Bros. Wii*.

2.3 Variance

Variance is a common statistical calculation that indicates the amount of dispersion between data points in a set [20]. Measuring variance in a dataset has numerous applications in statistics and probability, as well as in finance, investment, and machine learning. In these fields, variance is used to estimate the likelihood of an expected outcome. It can quantify the stability or volatility of a dataset and imply the existence of outliers.

The two primary types of variance are sample variance and population variance. Sample variance, as implied by its name, estimates the overall variation using a subset of the entire dataset. Population variance considers every data point in the set, providing an exact value for the amount of dispersion in the dataset. Variance, by definition, is the average of the squared differences of each data point from the mean.

A high variance indicates a large spread between data points and their mean. In the context of image processing, high variance indicates a high variety of colors in the image, while low variance means the image contains mostly similar colors. Figure 4 shows images taken from *New Super Mario Bros Wii*, where the tile outlined in green has a high variance, 934.91, and the one outlined in red has a much lower variance of 16.26. The variance of these two images was calculated on the RGB color space using the NumPy Python library.

2.4 Interpolation: Nearest Neighbor, Bilinear, Bicubic

There are two types of image upscaling: those that use solely the information contained in the image or video, and those that introduce outside data, generally from trained machine learning models. Methods that only use data contained within the image are generally more cost-effective in terms of resource utilization, as they require fewer and less intensive calculations, and there is no need to store additional data. One of these methods, interpolation, has three common variations: nearest neighbor, bilinear, and bicubic [21]. While these methods compute the upscaled image differently, all require the algorithm to be run on each channel of an image. Most images are stored in RGB format, meaning that the algorithm must be applied to the red, green, and blue channels of the image and then be recombined to create a single three-dimensional array of pixel values for the upscaled image.

Nearest neighbor is the simplest, both in terms of computational complexity and resource utilization. It involves iterating through the pixels of the upscaled image, identifying the pixel in the original image that is closest to the interpolated pixel, and copying the value of that pixel [22]. Outside of calculating which pixel in the original image is closest to the upscaled pixel, which is necessary for all forms of interpolation, the only action that is performed by hardware is the duplication of a value. While this algorithm is the most efficient, the results are fairly poor. As the level of upscaling increases, the resulting image becomes more pixelated. This approach may be suitable for some pixel art applications that require the exact shape and pixel distribution of the input (Figure 5), but it is not well-suited for applications that seek smooth and clear imagery (Figure 6).

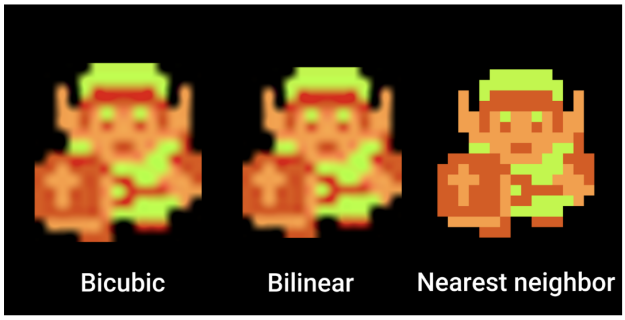

Figure 5: A comparison of bicubic, bilinear, and nearest neighbor interpolation and a pixel art character [4].

Figure 6: A comparison of nearest neighbor and bilinear interpolation on an upscaled picture of a flower [5].

Bilinear interpolation is an alternative to nearest neighbor [22]. Similar to the nearest neighbor method, it identifies the pixels in the original image that are closest to the upscaled pixel being calculated. However, instead of copying the nearest value, it performs a weighted average.

Bicubic interpolation is the third method, which employs a similar approach to weighting pixel values based on distance and summing the results, but with more data points [23]. Using bilinear or bicubic interpolation results in less pixelated images, as their calculations involve a series of averages; however, this approach leads to results that appear blurry (Figures 5 and 6) [22].

Unlike nearest neighbor, bilinear and bicubic calculations require more hardware resources. Although interpolation involves some division operations, the majority are addition, subtraction, and multiplication, which are fast on hardware due to the ability to parallelize elements of the calculation process. Additionally, given that pixel values in the RGB color space are 8-bit, it avoids computationally complex multiplications of large numbers. The difference in hardware resource usage depends on the number of input pixels accessed for each output pixel calculation. Bicubic interpolation requires four times the number of pixels that bilinear interpolation does, which increases the number of memory accesses-a common bottleneck in image processing. As such, when implementing these algorithms, special attention must be paid to how the input pixel values are stored.

All interpolation methods that use solely the information contained in the original image are subject to the introduction of artifacts, which must be handled with caution [24]. While some artifacts, such as those caused by overflow errors, can be mitigated through techniques like normalization, others, like pixelation, must be addressed by selecting a different algorithm.

To determine which algorithm best fits our goals and constraints, we created a table (Table 2) outlining the benefits and drawbacks of each method. By outlining the system's requirements and assigning a priority value to each requirement, the team determined that bilinear interpolation best met our needs. The highest priority requirement was achieving a result that wasn't pixelated, for it to blend with the convolutional neural network. Of the three methods, nearest neighbor was the only one that failed this criterion, and as such, we decided against it. The other two requirements were calculations and memory

Table 2: A comparison between three upscaling methods based on three requirements relevant to the high-level upscaling structure.

Interpolation Type	Number of calculations per pixel (low priority)	Number of memory accesses per pixel (medium priority)	Non-pixelated result? (high priority)
Nearest Neighbor	0	1	No
Bilinear Interpolation	19	4	Yes
Bicubic Interpolation	51	16	Yes

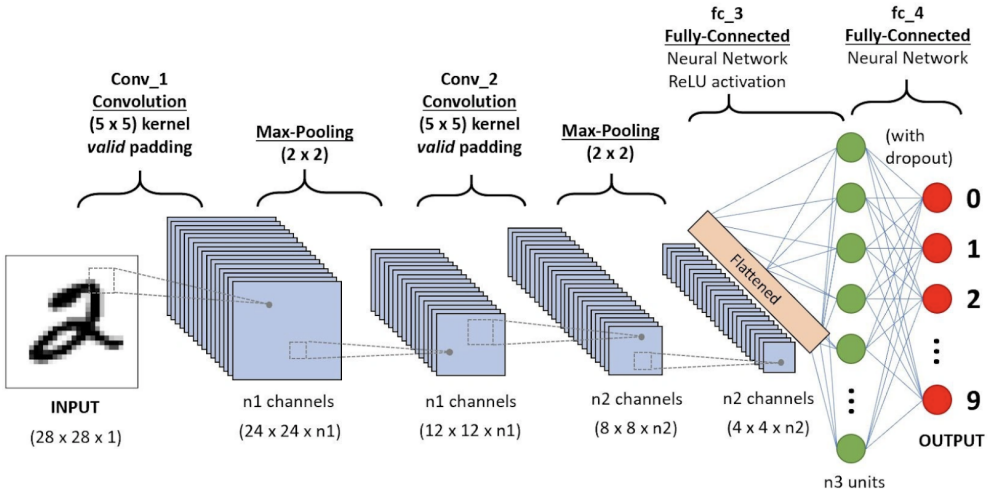

Figure 7: An example convolutional neural network [6].

accesses, with memory access ranked higher than calculations because there were more resources available for calculations than were available for memory storage. Additionally, memory usage posed a greater threat to speeding up calculations, as it limited the effectiveness of pipelining. However, regardless of the priority of the number of calculations and memory accesses required, bilinear interpolation uses fewer resources than bicubic interpolation.

2.5 Convolutional Neural Networks

The other set of methods surrounding image upscaling involves introducing external data. One of these methods, convolutional neural networks (CNNs), rely on performing convolutions on the input image using a series of pre-trained kernel weights. CNNs are broken into three main sections called layers: input, hidden, and output layers [25]. Input layers connect directly with the input data, hidden layers contain data calculation and manipulation, and output layers produce the results of the CNN. Regardless of the layer type, the output of each convolution layer is called a feature map.

When CNNs are used for image processing, only the input and output layers have the same format as the color space being used - the dimensions of hidden layers are often varied. The number of hidden layers varies depending on the structure of the CNN, and the function of each layer depends on the number of weights used and what traits the weights are trained to emphasize. Due to the detail contained in images, the number of weights used per hidden layer and the depth of the resulting feature map are generally larger than the number of color channels in the original image. While this allows for more detail in the upscaled image, it is one element of CNNs that requires more resources than the previously mentioned methods of upscaling. Figure 7 shows a typical architecture of a convolutional neural network.

Each layer of a CNN consists of multiply and accumulate steps, which combine input image pixel values with pre-trained kernel weights [26]. While there are multiple ways to convolve two 2D matrices, many image processing applications use matrix multiplication techniques or transposed convolutions. Convolving

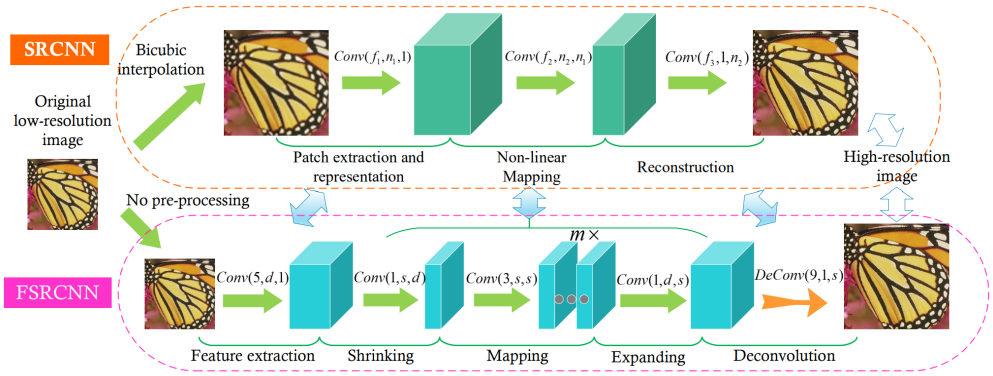

Figure 8: The differences between the Super-Resolution Convolutional Neural Network and the Fast Super-Resolution Convolutional Neural Network [1].

large quantities of data can utilize transformations into other domains, such as the signal domain, reducing the complexity of calculations. However, for CNN pipelines with a few layers or ones that take in relatively small-dimensional images, the process of converting domains can prove more computationally intensive than the resources saved by the conversion. Papers that detail real-time convolutional neural networks often discuss using vector-matrix multiplications [1][15][27]. Multiplications of small numbers are efficient on hardware, so matrix values and kernel weights can be implemented on an FPGA without a significant drain on resources.

Many neural networks use convolutional layers and fully connected layers [27]; convolutional layers combine feature maps with kernel weights through 2D convolutions, while fully connected layers map the results of convolution layers to a known dataset. This is often used to identify the features distinguished in the convolutional layers [28].

Fully connected layers are essential to CNNs, whose primary aim is image identification or facial recognition. Given that the goal of image upscaling is not to produce a description of what is being upscaled, but instead to increase the resolution of those features, many super-resolution CNNs are constructed mostly or entirely of convolutional layers. These layers identify important elements of the image that must be retained as resolution is increased, and they utilize the further application of trained weights to ensure the resulting quality matches that of images known to possess those qualities.

In addition to these layers, Rectified Linear Units (ReLUs) are applied as activation functions after a convolution to introduce non-linearity. ReLUs are necessary because they enable multiple layers to be used in a CNN, thereby creating a network [29]. They are necessary to introduce non-linearity to the network, given that convolution layers of matrix multiplications are solely linear operations. For a network to identify complex features, as required by a system that identifies a variety of features within different image frames, the network must include some level of separation between each convolutional layer. This separation ensures that each layer uses its convolution with feature weights to identify different elements within the image. Although there are many types of ReLUs, recent research notes the effectiveness of Parametric ReLUs (PReLUs) for super-resolution CNNs [15]. The purpose of a PReLU is still as an activation function, but their ability to learn makes them quicker to train with fewer resources [30].

Two well-known implementations of a CNN for super-resolution have been developed to increase an image's dimensions [1][14]. The first, Image Super-Resolution Using Deep Convolutional Networks, describes upscaling the input image using bicubic interpolation before the CNN is applied [14]. This method leverages the efficiency of interpolation to perform the change in dimensions and applies the increased resolution to the interpolated result, thereby helping to recover details lost when using interpolation alone. One downside of this method is that increasing the image dimensions at the start of the CNN means that all convolutional layers perform calculations on the larger image, resulting in more hardware being used for calculations and value storage. Another implementation, Accelerating the Super-Resolution Convolutional Neural Network,

addresses this downside by moving the step that increases the image's dimension to the end of the CNN with a deconvolution layer [1].

These two implementations also differ in the types of convolutional layers they use. SRCNN consists of only three convolutional layers: patch extraction and representation, non-linear mapping, and reconstruction. The second implementation, FSRCNN, has more resources to dedicate to the convolutional layers, given that each layer is smaller, allowing for more convolutional layers in total. As such, this implementation has layers for feature extraction, shrinking, multiple mapping steps, and expanding. The graphical representation of these methods is illustrated in Figure 8 [1].

2.6 Hardware Implementation of Upscaling Techniques

2.6.1 System Hardware

When implementing image upscaling on hardware, resource management and utilization are crucial considerations. Interpolation algorithms can have relatively small hardware footprints due to their minimal computational requirements. While there are some limitations when it comes to memory access, due to multiple calculations requiring the same pixel values from the original image, many of these limitations can be navigated by dictating memory access of certain calculations or storing multiple copies of an input image. However, hardware challenges rapidly increase when implementing a CNN. Not only do CNNs require many more calculations than interpolations do, but they also have higher memory usage due to the number of feature maps and weights that must be stored for each convolution layer. Complex CNNs can have millions of parameters and use billions of operations per input image [26]. Simpler CNNs, such as those detailed in [1] and [14], use significantly fewer resources, but still require careful resource allocation when implemented on hardware.

Many hardware implementations of image processors are carried out on FPGAs, given their versatility, rapid development time, and potential for high performance. However, FPGAs have limited resources for operations and memory storage [31], making the implementation process of a super-resolution CNN require careful consideration of resource allocation when determining the amount of calculations and the level of detail within each calculation. For CNNs that use weights that must be stored in external memory, high-precision values such as 32-bit floating-point numbers can cause bandwidth issues when loading weights into memory for calculations. In addition to the memory consumption, these high-precision numbers use more logic resources than their fixed-point counterparts.

Existing studies highlight how efficient use of FPGA resources can improve the quality or speed of an algorithm by up to 90% [31]. This resource utilization refers not only to bandwidth optimization for different data types but also to buffer management and simplifying logic. Implementers of CNNs should minimize external data access for values such as weights or feature maps by carefully handling data reuse. If done incorrectly, this can require reconfiguring the FPGA for different computation layers, which allows the reuse of some calculation resources, but drastically increases the time taken for upscaling. For super-resolution CNNs to achieve real-time implementation, this must be avoided.

2.6.2 Parallel Design Implementation

Research focusing on speed-ups within CNNs highlights the importance of parallel computation within and across convolution layers [26][31]. Parallel computation across layers focuses on performing multiple convolutions simultaneously, while parallelization within layers focuses on efficient loop unrolling and pipelining. Additionally, creating scalable CNN modules that allow the level of parallel computation to be quickly adjusted by changing a few high-level parameters enables tailoring resource usage to fit within different hardware constraints [26]. This is essential when working with different algorithms or across multiple FPGAs with different distributions of logic resources. Additionally, some research suggests implementing uniform loop unrolling factors across different convolutional layers [31]. This standardization ensures system-wide consistency, which is necessary when managing data flow in real-time.

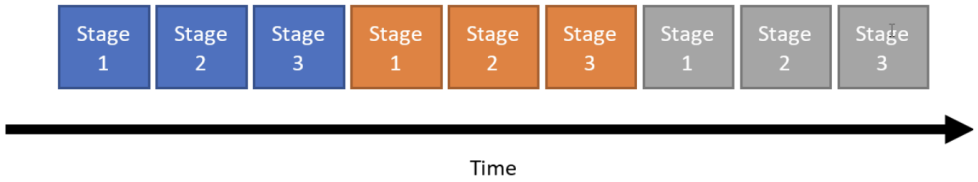

Figure 9: An illustration of a procedure with three processes that have no pipelining, so that the next process only starts after the first one is completed [7].

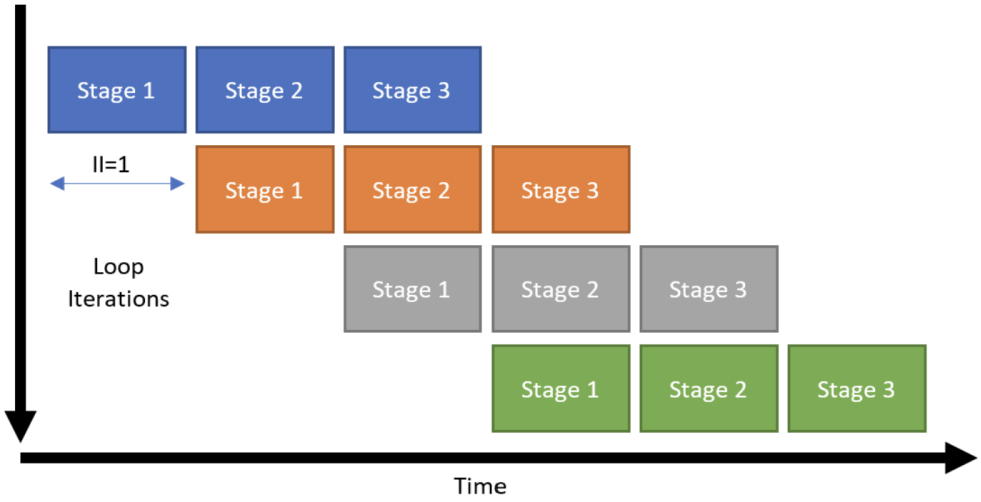

Figure 10: An illustration of a procedure with four processes that has pipelining with an initiation interval of one, so that the next process starts one clock cycle after the previous one is initiated [7].

One parallel implementation, pipelining, presents complications due to the potential overlap within processes. Pragma statements force the program to start another iteration of a process every specified number of clock cycles. A visual example of this can be seen in Figures 9 and 10, which highlight a process with three stages. Figure 9 illustrates an implementation without pipelining, where one process must fully complete before the next one begins. Figure 10 shows a version with an initiation interval of one. This means that a new process is initiated every cycle, resulting in overlap between processes. While specifying a low number can lead to heavy pipelining, it can drastically increase FPGA resource usage or cause complete timing failure.

If a system cannot meet the requested pipelining requirements, timing violations can occur. Some of these errors, such as initiation interval (II) violations, will cause a system to, for example, synthesize less effectively as it allocates resources for a timing goal that is unfeasible to achieve on the available hardware.

2.6.3 High-Level Synthesis

To work in hardware, a programming language or structure must be selected. While the standard is to use a hardware description language (HDL) such as Verilog, there are benefits to coding in C++ within the High-Level Synthesis (HLS) framework. Using Vitis HLS, C++ code is synthesised into a register transfer level (RTL) description. Similar to how working with FPGAs provides flexibility not available with Application-Specific Integrated Circuit (ASIC) design, working with HLS yields a quicker design phase than when coding directly in HDL. HLS abstracts many complex elements of HDL coding, and while HDL provides a finer level of control, implementing complex computation, such as a CNN, using HLS speeds up development time.

HLS addresses the challenges associated with hardware-level implementation through allowing con-

trol over synthesis and design optimization using pragma statements [32][33]. These control tools include resource allocation, binding, and scheduling, as well as control over the implementation of loops and streams regarding synthesis [34][35]. Memory control allows you to link specific variables or arrays with blocks of memory, such as URAM or BRAM. This level of control is essential when working on a board with varying amounts of memory resources or when the variables being stored require a specific type of memory access. The structure of HLS also contains several options for data control, specifically in relation to parallelizing processes [35]. Two such structures are for loop unrolling and pipelining, but both come with resource trade-offs. If implemented correctly, they allow for an overall processing speedup; however, they may require high quantities of logic resources, such as LUTs or DSPs, if the desired pipelining is aggressive [35]. Finally, beyond pragmas, systems of upscaling images using multiple stages will necessitate a method for passing values between functions or IP blocks, and HLS allows automatic creation and control of AXI4 protocols for this data transfer.

2.7 Verification and Universal Verification Methodology

Functional verification is a crucial step before hardware implementation because it ensures that an HLS implementation behaves exactly as its specification dictates. With modern digital designs containing millions of components, exhaustive formal proofs, and testing every possible scenario is impractical. Instead, verification relies on randomized, coverage-driven simulation to expose corner-case behaviour. The Universal Verification Methodology (UVM) is the industry standard framework for this task and plays a pivotal role. UVM packages a library of SystemVerilog base classes that encourage a layered architecture: sequences produce transactions, a driver converts those transactions into pin-level activity, a monitor observes the DUT's outputs non-intrusively, and a scoreboard compares them against a reference model [36]. The infrastructure, with its agents, environments, virtual interfaces, and configuration databases, is scalable and reusable, efficiently separating stimulus generation from protocol details and reference checking.

When using Vitis HLS, the device under test (DUT) is not in SystemVerilog but in C++. HLS comes with a built-in simulator, which compiles the DUT and a C++ testbench into an executable. Although this environment lacks UVM's libraries, the same concepts can be recreated in C++ with classes and helper functions. With C++, random stimulus corresponds to a sequence object that fills standard containers, such as streaming FIFOs. The driver is a small wrapper that packs the data into AXI-Stream stream beats and asserts handshake signals, the monitor is a symmetric unpacker, and the scoreboard is a comparator that reports mismatches and collects coverage metrics. HLS enables the parameterization of types within the C++ testbenches, allowing the same driver/monitor pair to be used for 8-, 32-, or 128-bit data widths, thereby achieving the interface reusability that UVM advocates. As such, the typical verification flow in HLS mirrors a UVM test. It compiles the DUT and instantiates a highly parameterized C++ testbench that embodies sequence, driver, monitor, and scoreboard roles. The testbench runs thousands of randomized iterations in C++ simulation and finally invokes Vitis HLS cosimulation to ensure that gate-level elaboration preserves the observed functional behavior [36].

The verification process employed the Monte Carlo method [37], generating thousands of randomized test inputs to simulate a wide range of scenarios and edge cases. Each test iteration used an independently sampled stimulus, allowing the testbench to explore the input space probabilistically rather than exhaustively. This approach gives strong statistical confidence that if no errors are observed across a large number of trials, the design will likely behave correctly in real-world use; when failures would otherwise occur, it has a very low probability [37].

In short, while the language and libraries differ, the methodology of UVM, being separation of concerns, transaction-level stimulus, and self-checking tests, maps cleanly onto HLS C++ testbenches, providing HLS designers with the same disciplined verification practices that have been long-established in conventional RTL development.

3 Design

This section provides an overview of how the team collaborated to develop the entire system. In this context, "design" provides a comprehensive explanation of how the team developed and implemented each system, where individual contributions overlapped, and how everything came together as a cohesive whole.

3.1 Functional Architecture

As mentioned in Section 1, the design of this project is centered on the architecture outlined in He et al. [15]. Since FPGAs have limited on-chip resources, the main idea is to split an image into subsections, or "tiles," and process them based on the variation of information they contain, as measured by their variance. If a tile has a high variance, it contains a lot of visual variation and should be upscaled via a high-quality algorithm. If a tile has low variance, for example, an image of a blue sky, it should be upscaled using a lower-quality but faster method. This split enables the high-quality method to be relatively computationally expensive while avoiding wasting clock cycles on unnecessary tiles. These two methods are FSRCNN, a convolutional neural network designed for super-resolution, and bilinear interpolation, a simpler image upscaling technique. This process is referred to as the "tile pipeline".

Our team developed a comprehensive high-level functional model that contained the entire system workflow. This functional model serves two purposes: validating the system and confirming the pipeline for hardware implementation, as well as verifying that the entire system functions correctly. The functional model outlined the complete pipeline, including cropping tiles, variance calculation, upscaling, repatching, and outputting to the video device. Additionally, the team used it to confirm that no tiles needed to overlap in the final system, ensuring proper upscaling without distortions between tiles. This entire process is illustrated in Figure 11, which shows the system used to upscale an image in C++ using the libtorch API.

The team computed variance with NumPy which helped to identify any potential issues that needed addressing and determine the appropriate variance threshold. Based on the variance calculation, the functional model sent the cropped image tile to either FSRCNN or interpolation for upscaling.

3.2 Model Training

Convolutional Neural Networks require trained weights for each convolutional stage, which are generated through machine learning. This machine learning implementation of super-resolution is a solved problem for modern datasets. If there is enough pixel density, creating pairs of images to train on is quite simple. On the other hand, upscaling retro video games presents a distinct challenge: style transfer. We chose a super-resolution model for upscaling that provides a low-operation, memory-efficient, and modular solution. We selected this as the top choice after examining [1], which demonstrated a successful real-time hardware upscaling system using FSRCNN. Developers typically create high-resolution and low-resolution image pairs for upscaling applications by downscaling the original image and then requiring the model to learn how to recreate it. The result usually produces an image of the same quality as the original, but with many times the number of pixels. Our application encountered several issues when implementing this approach. Additionally, modern image datasets have much higher quality than the raw Wii frames we worked with. The process of converting low-resolution frames back to their standard resolution counterparts did not yield the quality we had hoped for, which created a distinct challenge for our specific case.

This design utilizes a lightweight yet state-of-the-art model for image upscaling: FSRCNN. This model offers a balance between computational efficacy and the ability to enhance images suitable for an FPGA implementation. The team created low-resolution and high-resolution datasets from a Wii emulator before training the model. Various data augmentations, including random rotations, mirroring, and filters, are used to make the model as robust as possible on the available data.

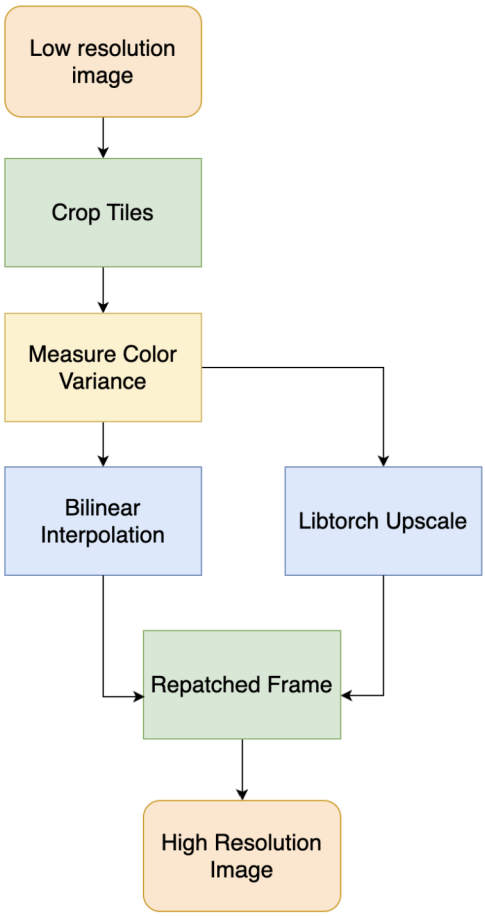

Figure 11: A block diagram of the C++ system architecture.

3.2.1 Machine Learning Design Plan

The objective of the machine learning (ML) component of this project is to upscale native Wii frames by a factor of two, aiming to improve pixel density, texture clarity, and overall image quality. As a reference for visual fidelity, the team utilizes output from the Dolphin emulator, which provides a modern, high-resolution rendering of the game. This serves as a benchmark for evaluating the quality of upscaled frames.

The initial approach involves generating training data by downscaling high-resolution frames captured from the Dolphin emulator and training a super-resolution model to reconstruct the original resolution. This method is preferred over using upscaled images as ground truth because models trained on artificially upscaled data are constrained by the limitations of the underlying interpolation algorithm. In contrast, training on downscaled-original pairs allows the model to learn meaningful representations and recover lost detail during upscaling.

The original dataset consists of 3,266 full frames captured from the emulator at a resolution of 1280 x 720 and a 16:9 aspect ratio. These frames are divided into 64 x 64 pixel patches, which are then downscaled to 32 x 32 to serve as low-resolution inputs. An FSRCNN model with 56 intermediate layers is trained on these 32 x 32/64 x 64 patch pairs to perform 2x image upscaling.

While initial results on emulator-derived data are promising, the model performs poorly when applied to frames captured directly from the Wii. The output exhibits limited improvement in pixel accuracy, texture clarity, and visual quality, indicating a mismatch between training data and real input conditions.

The team followed the design flow shown in Figure 12 throughout the model training process for

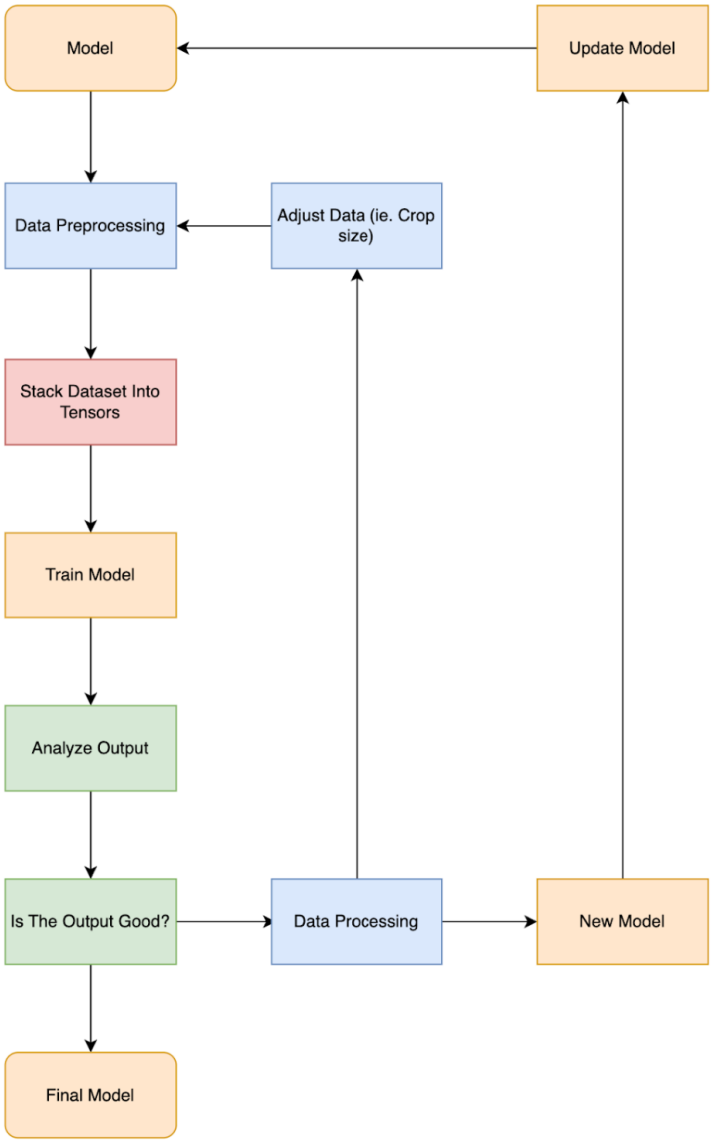

Figure 12: A flowchart detailing the process of determining next steps when training the model.

the initial FSRCNN for the emulated dataset.

3.2.2 Training on an Emulated Dataset with Composite Video Artifacts

To address this discrepancy, the dataset is augmented by introducing composite video artifacts into the low-resolution patches, simulating the visual characteristics of native Wii output. However, retraining the model on this augmented dataset still does not yield a substantial improvement in performance on actual Wii frames.

In response, the team constructed a new dataset consisting of 1,893 full frames captured directly from the Wii at 576i resolution with a 4:3 aspect ratio. Using the same preprocessing methodology, these frames are divided into 64 x 64 pixel patches and downscaled to 32 x 32 to generate the low- and high-resolution training pairs. This new dataset provides training data that is better aligned with the visual characteristics of native Wii footage, forming the basis for subsequent model training and evaluation.

While this new model provides an overall better upscaled image, satisfying one of the three project goals (accurate pixel upscaling), two major challenges remain to be addressed being texture clarity and overall image quality.

3.2.3 Addressing Domain Shift

Texture clarity and sharpness prove to be a challenge without a straightforward solution. This problem, known as domain shifting, requires a specialized approach as it lacks a standard implementation method. A more resource-intensive model can be used to create a high-resolution dataset for Wii frames by which FSRCNN can be trained on. To address this issue, an implementation of a CycleGAN model [38] was used. CycleGAN attempts to translate an image from a source domain to a target domain.

After looking into CycleGAN, a more modern successor was found, called Contrastive Learning for Unpaired Image-to-Image Translation (CUT) [39]. This model uses the same generator and discriminator architecture, making CycleGAN a good baseline. As CUT outperforms CycleGAN on most metrics, the team decides to implement this approach.

The team trained the CUT model on uncurated 256 x 256 pixel crops. Low-resolution Wii frames and high-resolution emulated frames are paired together so the model can learn how to transform Wii frames into the target domain of emulated frames. Due to the uncurated nature of the dataset, the results of CUT do not meet the team's quality standards.

3.2.4 Current Implementation

The final model weights were trained from the Wii dataset using the FSRCNN model. The frames are cropped into 64 x 64 pixel tiles, representing the high-resolution dataset. These tiles are then downscaled to 32 x 32 pixel tiles, representing the low-resolution dataset. Then, a variance calculation was run on both datasets to remove tiles that did not provide much information, such as sky tiles, black tiles, or any tiles that were completely the same color, as they lacked texture. This addressed one of the three issues (pixel density). The final architecture of the FSRCNN implementation for the board constrained a 7 x 7 kernel and 24 intermediate layers.

3.3 Managing Hardware Dataflow

The two major limitations on this project are time and resources. When testing the output video by simply passing it through the FPGA without any manipulation, the output FPS was 25. This number sets the target for what must be achieved for the design to qualify as a real-time system. Each frame, 414,720 pixels, has to be loaded into the system, split into 32 x 32 tiles, variance calculated, converted to RGB, upscaled using interpolation or FSRCNN, reassembled into the upscaled frame, and output to a frame buffer within 40 milliseconds before the next frame arrives. In addition to timing constraints, hardware limitations also narrow the capabilities of the entire system. When considering the resources of the ZCU102, the larger and more robust board the team worked with, it has a total of 1,824 BRAM blocks, 2,520 DSPs, 5,481,600 FFs, and 2,740,800 LUTs available to the user. Ensuring that every IP block created could fit within these constraints was one of the project's largest challenges.

The team first designed the architecture to target the KV260 development board, an entry-level evaluation platform for the Zynq Ultrascale+ series SoCs, as it was an affordable and seemingly suitable option at the project's inception. Later, the team received a more capable ZCU102 development board and ported all the progress we had made up to that point onto it. All IP was to interface with each other via AXI-Stream, a high-bandwidth interconnect system designed by ARM, and operate on 28 x 28 pixel tiles [40]. Later, the team switched to 32 x 32 pixels to avoid dealing with partial stream transfers. To maximize bandwidth and minimize the data-streaming bottleneck, all IP blocks were developed to be data-driven and have a top-level interface stream width of 128 bits.

Since each IP block expected data to arrive as 32 x 32 tiles in a raster format, but image frames are stored in DRAM linearly, the design required a way to reformat the data. One option was to store the image frame in on-chip Block RAM (BRAM), one block per row of pixels, and to simply read 32 pixels from the first RAM, 32 from the second, and so on, until a full tile was read, but this exceeded the resources available on the KV260. The second option, and the one the team used, was to instead send tiles down one row at a time using an AXI-DMA core in scatter-gather mode to fetch the relevant sections of an image. This, however, created a bottleneck. With a 128-bit stream interface, two bytes per YUV 4:2:2 pixel, and 32 pixels per row in a tile, only four stream beats of data could be output before the DMA engine needed to start a new transfer. This ended up adding a significant amount of latency to the design, since the DMA engine would need to access DRAM to fetch a new buffer descriptor every four stream beats. A single DRAM chip was shared with PetaLinux, indicating a busy memory interface.

The format of the AXI-Stream interface between IP blocks in the tile pipeline is the following: {0BGR-0BGR-0BGR-0BGR}. If the RGB values were sent out without padding, each AXI-Stream transfer would contain 5.333 pixels, making storage and indexing of pixels down the pipeline much more difficult. The team opted to simplify code and calculations at the expense of slightly more memory. This could be further optimized in the future, making every AXI-Stream contain only valid pixel data and no padding. However, handling the pixels would be much more difficult and wouldn't be worth the effort for the timeline of this project.

One additional piece of hardware at the end of the pipeline is a block that converts RGB 8:8:8 to RGB 5:6:5. Most of the system operates in the RGB 8:8:8 color space to preserve a higher level of detail, but the frame buffer of the PetaLinux installation on the KV260 and ZCU102 expects data in RGB 5:6:5 format. A simple IP in Verilog clips the least significant bits from each color channel and repacks the data. For each pixel, R and B are clipped to remove the 3 least significant bits, and G has its 2 least significant bits removed. This results in a final AXI-Stream width of 64 bits sent into the SG-DMA, since half of the bits from the 32-bit RGB 8:8:8 are removed.

3.4 High-Level Synthesis Development

A consistent design methodology is employed across all blocks developed in HLS, as illustrated in the flowchart (Figure 13). To begin, the team created an example stimulus for each block to use. This could be used by self-tests and verification implementations to confirm the accuracy of the intended result. While it isn't a replacement for extensive testing, it is foundational to the initial development of upscaling blocks since it illustrates the visual result of a successfully upscaled tile and provides known values to reference when manually debugging.

While the ideal reference is created, each IP's architecture is defined. This results in high-level block diagrams that illustrate the general inputs, outputs, and internal features of each block. Not only is this essential for creating a development framework, but it also provides an understanding of how data should be transferred to and from a block, allowing verification tests to accurately build these connections.

After the general architecture was defined, the computations required were analyzed. If the expected necessary calculations for a given block are deemed trivial, implementation would start immediately in HLS; however, if the block is complex, a functional model would first be created in Python. This functional model enables the team to test whether the architecture produces the expected output, and if not, to highlight the need to debug the Python model or reevaluate the validity of the proposed architecture. If the functional model correctly calculates the expected input, implementation could begin in HLS using the structure of the functional model as a framework. Creating functional models in Python can save a lot of time in the design process, as it eliminates high-level bugs in an architecture before starting the longer process of implementing a design in HLS.

Once a block is created in HLS, it is verified against an extensive testbench to confirm that its implementation matches the expected outcome. These testbenches also enable the cosimulation of the block, providing waveforms of the IP's input and output signals. These signals are essential for confirming that the block interfaces correctly with the overall system and can be used to calculate the block's process time.

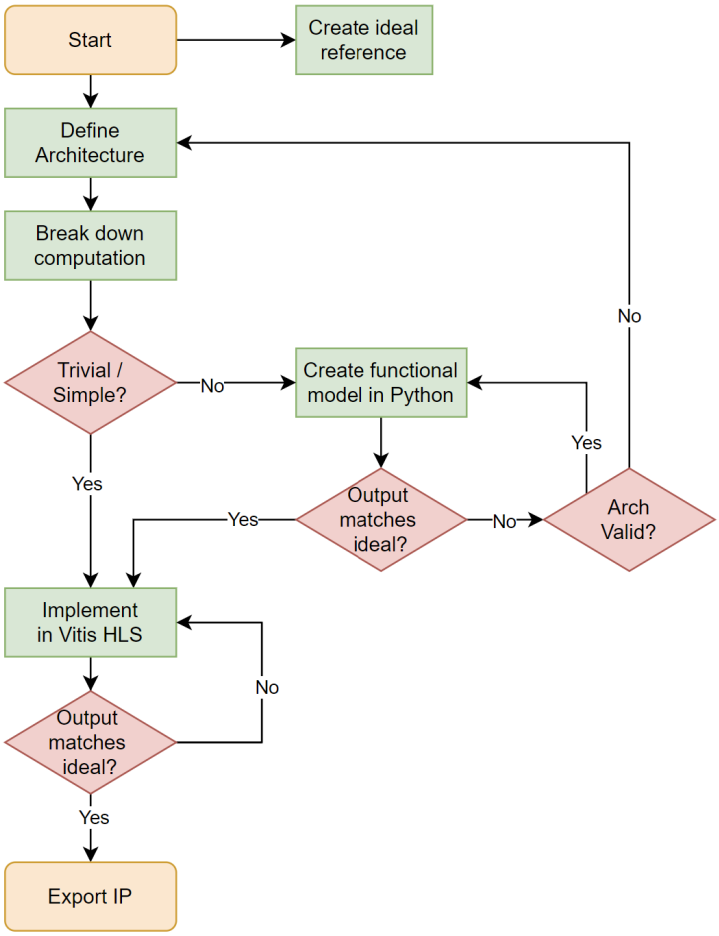

Figure 13: A flowchart illustrating the development process for IP blocks using HLS.

3.4.1 Variance and Color Space Conversion

3.4.1.1 Functionality

The first block used in the tile pipeline determines the variance of pixels within a tile and converts the values from YUV to RGB. These functions were combined into a single block given their simplicity and interconnectedness. When the block's architecture was defined and analyzed, the team determined it to be a simple implementation. Because of this, it was unnecessary to create a functional model in Python, and development started directly in HLS. Variance functionality was essential to the project as it helps to determine which algorithm will be used to upscale any given tile.

The native color space of the Wii video is YUV 4:2:2, while the PetaLinux frame buffer operates on RGB 5:6:5, creating a need to convert color spaces at some point within the system. Once the input and output color spaces were determined, a decision had to be made about the intermediate color space conversions and how to retain as much image data as possible during said conversions. Since the RGB color space is easier to work with, more familiar to the team, and the final required output of the system, it made the most sense to convert from YUV to RGB early in the tile pipeline, allowing most of the computation to take place in the RGB color space. Given this, the team chose to convert from YUV 4:2:2 to RGB 8:8:8 after the variance calculation - so it could be computed simply using the Y channel - but before any upscaling occurred.

Each 32 x 32 pixel tile of each video frame is first sent through the variance and color space

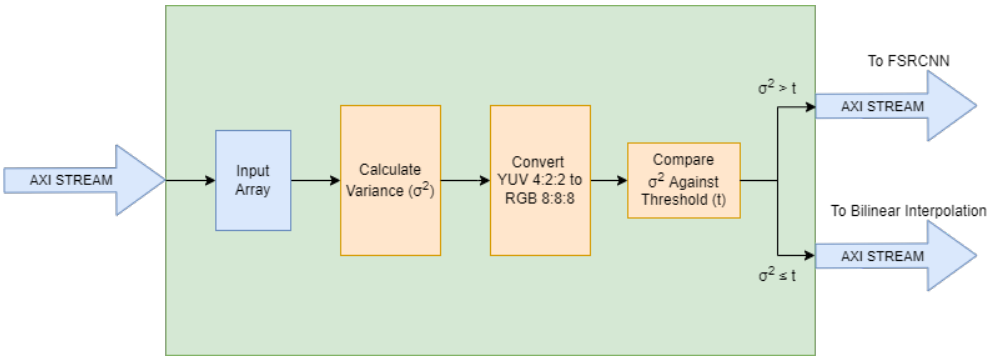

Figure 14: A block diagram of the variance and color space conversion IP.

conversion block. To reduce the chance of error and make debugging easier, these two pieces of code were separated into two functions within the same block, and separate testbenches were written to identify issues within each part of the code. The first iteration of the code consisted of only writing and testing the variance calculation on a tile with a top-level 32-bit AXI-Stream width. After this was verified to work properly with self-tests, the color space conversion code was developed. During this process, the team decided to change to a 128-bit AXI-Stream width, which required slight adjustments to the way internal arrays were indexed to handle larger pulses of information entering the block. Due to the slightly more complex nature of converting color space compared to calculating variance, more intensive verification tests were written to ensure the code functioned properly, in addition to the simple self-tests that were already in place. The block diagram of the whole IP can be seen in Figure 14.

3.4.1.2 Variance Calculation Implementation

The beginning of the variance IP starts with a loop that reads AXI-Stream data into a buffer. Arrays were chosen for buffering because they allow for accessing certain pixels multiple times throughout the code. The input array, which holds all the YUV pixels, has a depth of 128, with each index containing a 128-bit integer that stores 8 YUV pixels. At the output, the array was required to be twice the size of the input array, due to reasons that will be elaborated upon later.

One additional input to the block is an override controller register, set via AXI Lite. This is used as a sort of debug switch for hardware that allows the whole tile to skip the variance calculation and be sent directly to the color space conversion, and out to one of the upscaling methods. This was very helpful when the block was being tested in tandem with only one upscaling block, without the other, to ensure every tile goes through the desired interface.

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{N} \tag{1}$$

Analyzing the formula for variance calculation (Equation 1) reveals that it must be split into two separate loops: the first to calculate the mean of all data points (μ), and the second to compute the sum of every squared difference of luminance and the mean (σ^2). The first loop iterates through each 128-bit transfer, indexing just the luminance (Y value) from each pixel to sample for the calculation. Since YUV 4:2:2 only samples the Y channel for every pixel, and the U and V values are shared for every 2 pixels, computing the variance on only the luminance of the image gets a sample of every pixel that accurately represents the image contents while saving resources by only computing on one channel. As each Y component is isolated, they are all added together to find the sum, then divided by the total number of pixels in a tile to find the mean luminance value of the tile. The second loop indexes the Y values in the same manner and finds their difference from the mean luminance. The differences are then squared and added to a variance sum, which is again divided by the number of pixels in a tile to find the variance for the tile. Self-tests were written specifically for the variance calculation, which passed non-random values through and checked the calculated variance against the expected, hand-calculated value. Once the variance calculation was verified to work

correctly, development of the color space converter began.

3.4.1.3 Color Space Conversion Implementation

The entire color space conversion calculation is performed in a single loop that iterates through every YUV pixel in a tile, converts them to RGB, and packs the results into an array that is twice the size of the input array. To convert the YUV 4:2:2 values into RGB 8:8:8, each 128-bit data stream needs to be split up into 32 bits of Y0, U, Y1, and V values. Two pixels of RGB data are to be extracted from every 32 bits of YUV input data, equalling 64 bits after padding. Equations (2), (3), and (4) show the floating-point conversions for YUV to the R, G, and B color channels, respectively [41].

$$R = Y + 1.402 * (V - 128) \tag{2}$$

$$G = Y - 0.344 * (U - 128) - 0.714 * (V - 128) \tag{3}$$

$$B = Y + 1.772(U - 128) \tag{4}$$

The first iteration of the design utilized these formulas as they were, but an excessive amount of resources were being used for computation, and the calculation was also slower than ideal. A previous research paper proposes [41] the potential for converting these formulas into bit shifts to increase efficiency; these ideas were used to change the original formulas into the equations used in the final design, (5), (6), and (7). Manipulating the floating-point multiplications into equivalent equations using constant integer multiplications with bit shifts significantly reduced the resource utilization of the block and accelerated the calculations.

$$R = Y + ((1436(V - 128) + 512) >> 10) \tag{5}$$

$$G = Y - (352(U - 128) + 512) >> 10) - ((732(V - 128 + 512) >> 10) \tag{6}$$

$$B = Y + (1436(V - 128) + 512) >> 10) \tag{7}$$

After the multiplications, the RGB values must be clamped to the range of 0-255 to ensure they remain within the requirements of an 8-bit format. Without this, simulation testing revealed overflow and wrapping errors, where some pixel components were significantly off from the expected value. Clamping the values was necessary to keep them within the desired domain without encountering any overflow issues. Another similar problem arose from the definition of the fixed-point data type that was used in the code. The default settings in HLS for an arbitrary precision fixed-point are AP_TRN for quantization and AP_WRAP for overflow, which truncate to minus infinity and wrap around, respectively [34]. These settings cause confusing errors when many expected output values of 255 were returned as 0 from the function. Changing these settings to AP_RND and AP_SAT - round to plus infinity and saturate - fixed this bug and made the results much more accurate. The output of this block was also verified by implementing the design in hardware and simply viewing the video, ensuring that the output appears exactly as it did before being converted to RGB.

After the color space conversion function is called from the main function, the output logic sends the RGB data out via the AXI-Stream interface. If the override mode is set to convolution or interpolation, the IP sends all tiles out without computing the variance. Otherwise, the variance calculation for each tile is compared against a threshold set by AXI-Lite. If the variance is higher than the threshold, the tile is sent out the convolution AXI-Stream output; if it is less than or equal to, it is sent to interpolation. This ensures that low-variance tiles are upscaled using the faster, simpler method, while tiles that require more intense upsampling are handled accordingly.

3.4.1.4 Implementation Results

One 32 x 32 pixel tile is sent through the variance and color space conversion block in 6.5µs when cosimulated in HLS. When applied to an entire frame, this IP takes 2.4 ms to complete. With a target frame rate of 25 FPS, this block consumes 5.98% of the 40 ms allowed for the entire system to achieve real-time video

upscaling. Synthesis in Vitis HLS reports that the design uses 11 BRAM blocks, 16 DSPs, 955 FF, and 7018 LUTs. Each of the first three metrics is negligible when compared to the total resources on the ZCU102; LUTs take up approximately 2% of the total resources on the board.

Table 3 shows the resource utilization difference between floating-point and constant integer multiplications. After changing the equations from (2), (3), and (4) to (5), (6), and (7), the DSPs, FFs, and LUTs all drastically reduced their utilization.

Table 3: Resource utilization comparison between floating-point and constant integer multiplications for the ZCU102 board.

Multiplication Type	**BRAM**	**DSP**	**FF**	**LUT**
Float	11 (0.6%)	272 (10.8%)	26357 (4.8%)	71954 (26.3%)
Constant Integer	11 (0.6%)	16 (0.6%)	955 (0.2%)	7018 (2.6%)
Total	1824	2520	548160	274080

3.4.2 Bilinear Interpolation

3.4.2.1 Hardware Architecture

We use bilinear interpolation as a cheap upscaling method for the tile. It produces a happy medium quality result while accessing only four input pixels per output pixel. Although multiple pixels from the upscaled image require the same low-resolution pixel values for their computation, careful unrolling and pipelining can be done to avoid memory access issues.

When upscaling using this method, the IP iterates through the pixels in the output tile and identifies the four nearest points from the input tile. It performs a weighted average on these points by identifying their distance to the upscaled point, and the weight is found by dividing this distance by the total distance between points. It then multiplies the weights by the pixel values and sums them. This resulting sum is the value of the upscaled pixel. Equation (8) demonstrates the equation used, and Figure 15 provides a graphical representation of the low-resolution points in relation to the one being calculated.

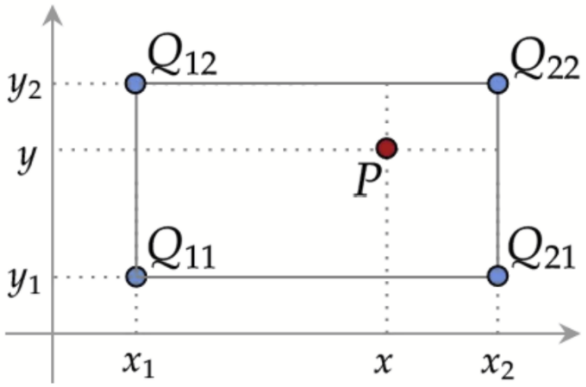

Figure 15: A graphical representation illustrating the values used in Equation (8) [8].

$$P = \sum_{i=1}^{2} \sum_{j=1}^{2} \frac{(x_{3-i} - x)(y_{3-j} - y)}{(x_2 - x_1)(y_2 - y_1)} Q_{ij} \tag{8}$$

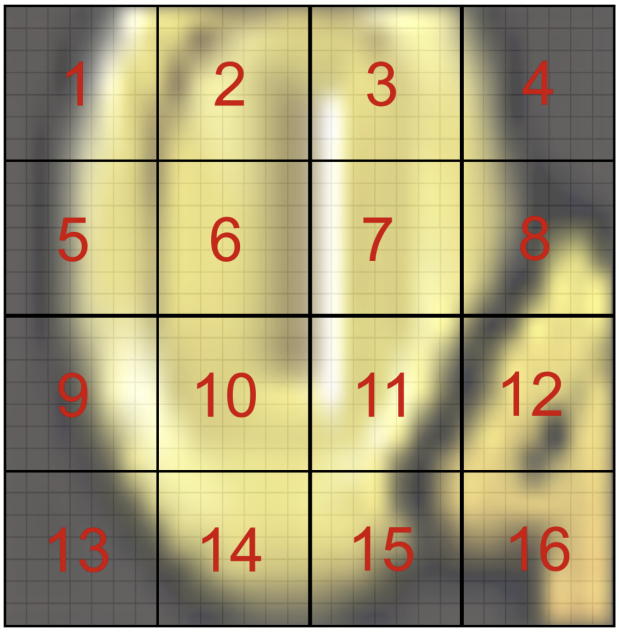

Figure 16: The image from the coin tile test broken into 16 7 x 7 pixel sections as is done in the bilinear interpolation block.

3.4.2.2 Functional Modeling

Based on the HLS design flowchart (Figure 13), bilinear interpolation is non-trivial, so it requires a Python functional model. This functional model emulates many of the functions required by the HLS implementation, the first of which is the calculations used. The bilinear interpolation functional model utilized an upscaled result calculated using a standard library to verify the numerical accuracy of the model.

The Python functional model also upscales subsections of the tile separately and then stitches them back together into a fully upscaled tile. Similar to how the full frame is tiled in 32 x 32 sections, the bilinear interpolation block separates each tile into square sliders, which are upscaled concurrently. When breaking the image into sliders, an overlap between sections needs to be accounted for. Without considering an overlap, pixels on the borders of the slider sections are incorrectly upscaled due to the lack of neighboring pixels. Figure 16 illustrates how the ideal test tile is split into sections, and Figure 3.7 highlights the overlap between sections. While this use of slices didn't change anything about the implementation in Python, this upscaling via smaller sections is necessary for HLS development. Although each section can be passed to the bilinear interpolation function separately, full rows of sections are sent into the function due to the nature of the HLS implementation. For this method, sections 1, 2, 3, and 4 in Figure 17 are passed to the calculation function as a single matrix. This design is used because the HLS implementation's AXI-Streams pass in pixels row by row, so most of sections 2, 3, and 4 are filled before section 1 contains the necessary values to begin calculations. The time required to load the final row of tiles for the extra sections to complete the row of sections is negligible.

The creation of the HLS implementation is based on a passthrough FIFO, which reads pixel values from an AXI-Stream, stores them in a FIFO, and then passes the values out through another AXI-Stream. This is essential for ensuring that pixel values can be parsed from the stream's data, manipulated as an integer, and then transformed back into the stream's data type. This process provides the foundation for the bilinear interpolation block, as it builds the shell of an HLS block that can be successfully synthesized into RTL and implemented in the overall system. It also confirms that real-time image passthrough was possible using HLS on the hardware provided.

This process highlighted issues with the AXI-Stream protocol that were not apparent in the back-

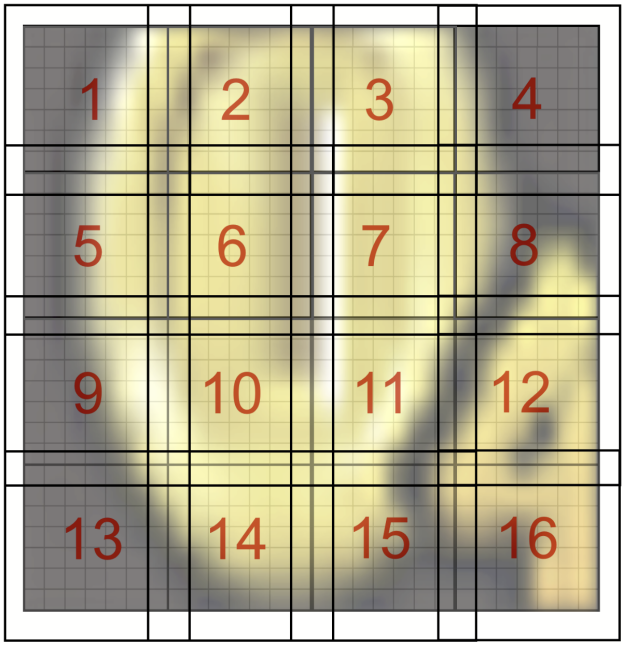

Figure 17: The image from the coin tile test broken into 16 7x7 pixel chunks illustrating the chunk overlap necessary for successful interpolation.

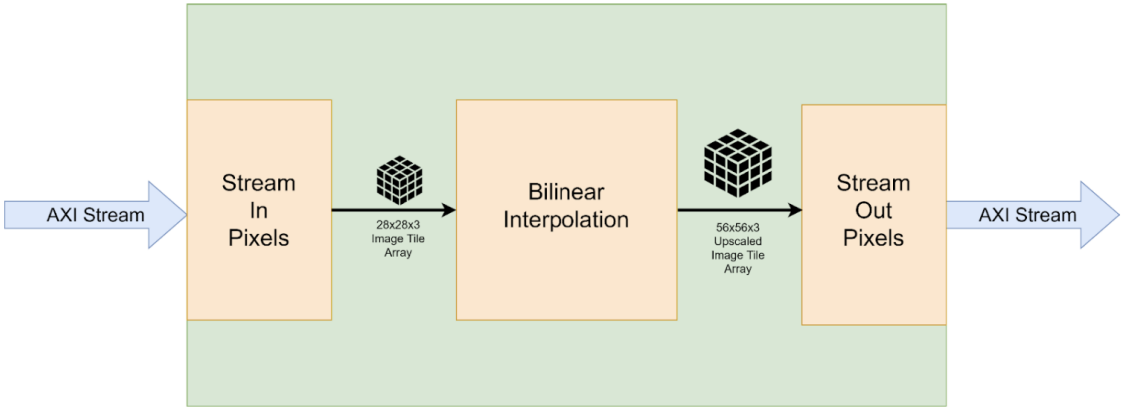

Figure 18: A block diagram illustrating the structure first implementation of the bilinear interpolation IP in HLS.

ground research. While the team expected using HLS to implement the AXI-Stream protocol to be straightforward, more default signals were created than originally expected, according to the AXI protocol documentation. It is possible to disable some of these signals; however, the most recent HLS documentation did not mention the existence of the signals or how to remove them. This feature is only available in Vitis HLS 2024.1, but the team used version 2023.1. As such, identifying their existence before the implementation of calculations began allowed the team to narrow down the possible causes of the additional signals to effectively disable them. Some of the additional signals, specifically TKEEP and TSTRB, could not be disabled and were therefore set to remain high to prevent interference with the DMA processing of the stream.

Figure 18 shows the block diagram of the first HLS implementation. This implementation requires the entire image to begin calculations and uses values stored in 3D matrices. Although this was later updated, the initial implementation utilizes 8-bit AXI-Stream transfers and performs a 2x upsampling on a 28 x 28 x 3 input.

```
for (int ch = 0; ch < CHANNELS; ++ch) {

    #pragma HLS UNROLL factor=3

    float interpolated_value =
        w00 * image_in[y0][x0][ch] +
        w10 * image_in[y0][x1][ch] +
        w01 * image_in[y1][x0][ch]] +
        w11 * image_in[y1][x1][ch]];

    // Assign the interpolated value to the output image
    int out_index = (y_out * WIDTH_OUT + x_out) * CHANNELS + ch;
    image_out[out_index] = static_cast<pixel_t>(std::round(interpolated_value));
}
```

Figure 19: A code snippet showing the process of interpolating a pixel with unroll statement.

The first interpolation implementation, with no optimizations, yields a calculation time of 376 µs and a tile process time of 496 µs. Given that each frame at the time is made up of 416 28x28 tiles, this speed would yield a frame rate of 5 FPS, significantly lower than the required 25 FPS for a real-time implementation.

Next, optimizations are added in the form of HLS pragma statements. The two used for speed-up are HLS UNROLL and HLS PIPELINE. Loop unrolling statements are used within the calculation iterations to force the synthesis tool to implement multiple iterations of the loop concurrently. The value specified in the pragma is the number of loop iterations happening simultaneously, so when upscaling each color channel of a pixel, the innermost loop is unrolled by a factor of three. Loop unrolling involves the potential risk of overlapping memory accesses of the same value within one clock cycle, but this is avoided in color channel unrolling since each of the color values is stored separately (see Figure 19)

Pipelining was also added to the bilinear calculations in the uppermost loop that iterates through the rows of the tile being upscaled. We attempted to pipeline it so that a new row starts being upscaled every 11 clock cycles, but this lead to II violations. Instead, we used a pipelining level of 76, which is the smallest pipelining value we could achieve without causing violations.

Without altering the code structure, implementing these pragma statements reduced the total calculation time to 137 µs per tile, resulting in a total process time of 275 µs. While this speedup increased the bilinear interpolation to 9 FPS, it was still below the expected value of 25, needed for real-time.

Next, we increased the AXI-Stream transfer width from 8 bits to 128 bits, which uses fewer stream beats to pass a full tile into the IP block. Additionally, the data type of stored pixel values and intermediate calculation variables was changed from floating-point to fixed-point to reduce logic resource utilization. The first implementation of bilinear interpolation used floating-point variables, which require approximately 75,500 LUTs, due to the additional memory and logic required for floating-point calculations. Floating-point calculations are more accurate than fixed-point calculations, but given the interpolation block's goal of providing a low-resource upscaling method, using a less accurate data type that consumes fewer resources is preferred. To reduce logic usage, floating-point variables were updated to 32-bit fixed-point representation with 12 fractional and 20 integer bits. This decreased the LUT usage to approximately 48,600, but increased DSP usage from 47 to 885, indicating that further work was needed to reduce resource usage.

The last major change in this block was restructuring the HLS to match the layout described in the Python functional model (Figure 20). Instead of the data from AXI-Stream transfers being directly parsed into the matrix for upscaling calculations, an intermediate process was implemented that stores the

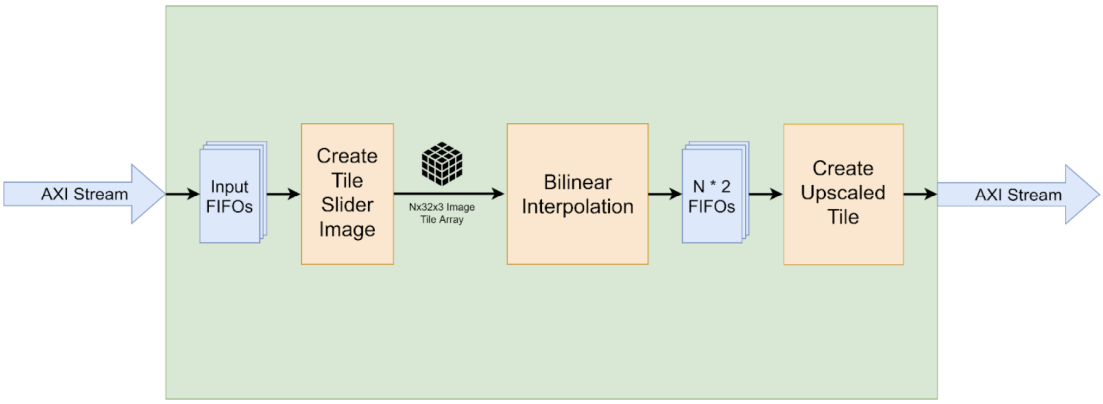

Figure 20: A block diagram illustrating the structure of the final implementation of the bilinear interpolation in HLS.

data packets in an internal FIFO. A separate function reads this FIFO and parses the data into individual pixel values. Next, the separated pixel values are stored in a matrix to be upscaled by the same bilinear interpolation calculation function used in previous iterations. The upscaled pixel values are transferred out of the calculation function using FIFOs, then combined into the upscaled version of the tile to be transferred out via AXI-Stream.

The implementation of these changes not only speeds up the tile processing time but also significantly reduces the amount of resources used. These updates yield an implementation that utilizes only 40 DSPs, approximately 17,500 LUTs, and performs the upscaling of a 28 x 28 tile in 50 µs.

Once the final version of the implementation was created, several parameters were adjusted to understand how changing the internal sliding window size affects the resource utilization and process time. The initial iterations of the bilinear interpolation use 7 x 28 pixel tiles, but switching to 4 x 32 pixel sections to account for 32 x 32 pixel tiles results in a slight increase in resource consumption, using 46 DSPs and roughly 18,500 LUTs, while timing remains the same. We expected that increasing the tile size would increase resource utilization, but we anticipated a slight decrease in computation time because more of the block would be parallelized. Given the minimal increase in resource utilization, the team surmises that this could be attributed to a relatively small increase in parallelization, as the number of sliders is only doubled from four to eight. Additionally, although computation is a significant part of the overall tile process time, streaming out upscaled tile values still accounts for a substantial percentage of the time, which remains unchanged with this update.

3.4.2.3 Implementation Results

The final iteration of the bilinear interpolation IP, with all implemented speed-ups, yields a process time of 62 µs on a 32 x 32 pixel tile when cosimulated in Vitis HLS. Multiplying this by the number of tiles in a full image, 368, yields an image process time of 0.02 seconds or 43 FPS. This value is compared with all other bilinear interpolation iterations in Table 4. However, when measured by the ILA, this number jumps to 290 µs, yielding an FPS of around 10. However, when this is combined with the system as a whole, the DMA's data transfer speeds also prove to be a limiting factor. The difference between the cosimulation waveform and the ILA waveform should be further studied to understand the underlying cause of the discrepancy in their values. This implementation uses 89 BRAMs, 40 DSPs, and roughly 18,500 LUTs. When comparing with the resources available on the ZCU102, it is a low percentage of the available BRAMs, DSPs, and LUTs, with them using 4.8%, 1.8%, and 6.7% of these resources, respectively. This shows that the interpolation block achieved the goal of providing a fast upscaling method with low resource utilization.

Table 4: Resource utilization comparison between each iteration of the bilinear interpolation block on the ZCU102. The process speed measurement is taken from the co-simulation waveforms in Vitis HLS.

Iteration	BRAM	DSP	LUT	Process Speed
No Speed-ups	10 (0.5%)	47 (1.8%)	75,500 (27.5%)	496 µs
Pragmas Introduced	10 (0.5%)	47 (1.8%)	75,500 (27.5%)	275 µs
Switch from Fixed-Point to Floating-Point	10 (0.5%)	885 (35%)	48,600 (17.7%)	275 µs
Sliders and FIFOs Introduced	89 (4.8%)	40 (1.6%)	17,500 (6.4%)	62 µs
Tiles Changed from 32×32	89 (4.8%)	46 (1.8%)	18,500 (6.7%)	62 µs
Total	1824	2520	274080	—

3.4.3 FSRCNN

As discussed previously, the project uses FSRCNN as the high-quality upscaling method. Development began later than anticipated due to delays in earlier design stages. The architecture design of our implementation focuses on minimizing resource utilization while maximizing throughput, leveraging two key HLS design patterns: dataflow and pipelining.

3.4.3.1 Hardware Architecture

The primary bottleneck in implementing convolution algorithms on hardware is the high demand for memory access; for example, a 3 x 3 kernel requires accessing the same pixel up to nine times. Conventional implementations typically rely on a sliding window approach, which computes one element of a feature map per unit time. However, this method tends to over-instantiate and under-utilize Block RAM (BRAM) resources.

To address this, the team designed the system around a FIFO-based architecture, originally proposed by Panchbhaiyye et al. [9]. This architecture processes one row of an input channel at a time in raster order and computes the final multiply-accumulate (MAC) sum of the kernel and window incrementally. Figure 21 illustrates the architecture in detail.

As an example, the first row of the kernel is MAC'd with row i, and the resulting sum is queued into a FIFO storing partial sums. When row j comes around, the first partial sum is dequeued and added to the MAC result of row j and the second row of the kernel, which is then enqueued in a second FIFO. The first row of the kernel is concurrently MAC'd with row j and enqueued back into the first FIFO. Once row k starts being read, the MAC result of row k and the third row of the kernel is added to the dequeued value from the second FIFO, summed with the bias term, and passed through an activation function. Each set of these partial sum FIFO-MAC is referred to as a "processing element" (PE), and can compute the output for one feature map of the convolution operation at a time.

This architecture allows for each input channel to be flattened and stored in a BRAM (Figure 22). A single BRAM is instantiated for each input channel, enabling read operations to be executed completely in parallel across channels. The BRAMs can also be implemented as FIFOs, allowing the next layer of the network to start computation before the previous layer is done under the dataflow pragma (Figure 23) [34]. Partial sum FIFOs are also implemented using BRAMs.

3.4.3.2 Functional Modeling

The FSRCNN IP block follows the HLS flowchart shown in Figure 13. Because the architecture is non-trivial, the team takes the Python functional-model route. To this end, the team developed a functional model of a single convolution layer; this is the atomic unit in the neural network and thus serves as a foundation for further development. Figure 24 outlines the high-level development process, beginning with the creation of an ideal reference that the team provides to all three environments: Vitis HLS simulation, the functional model, and the ideal PyTorch result. To ensure the Vitis HLS model is valid at a high level, the team

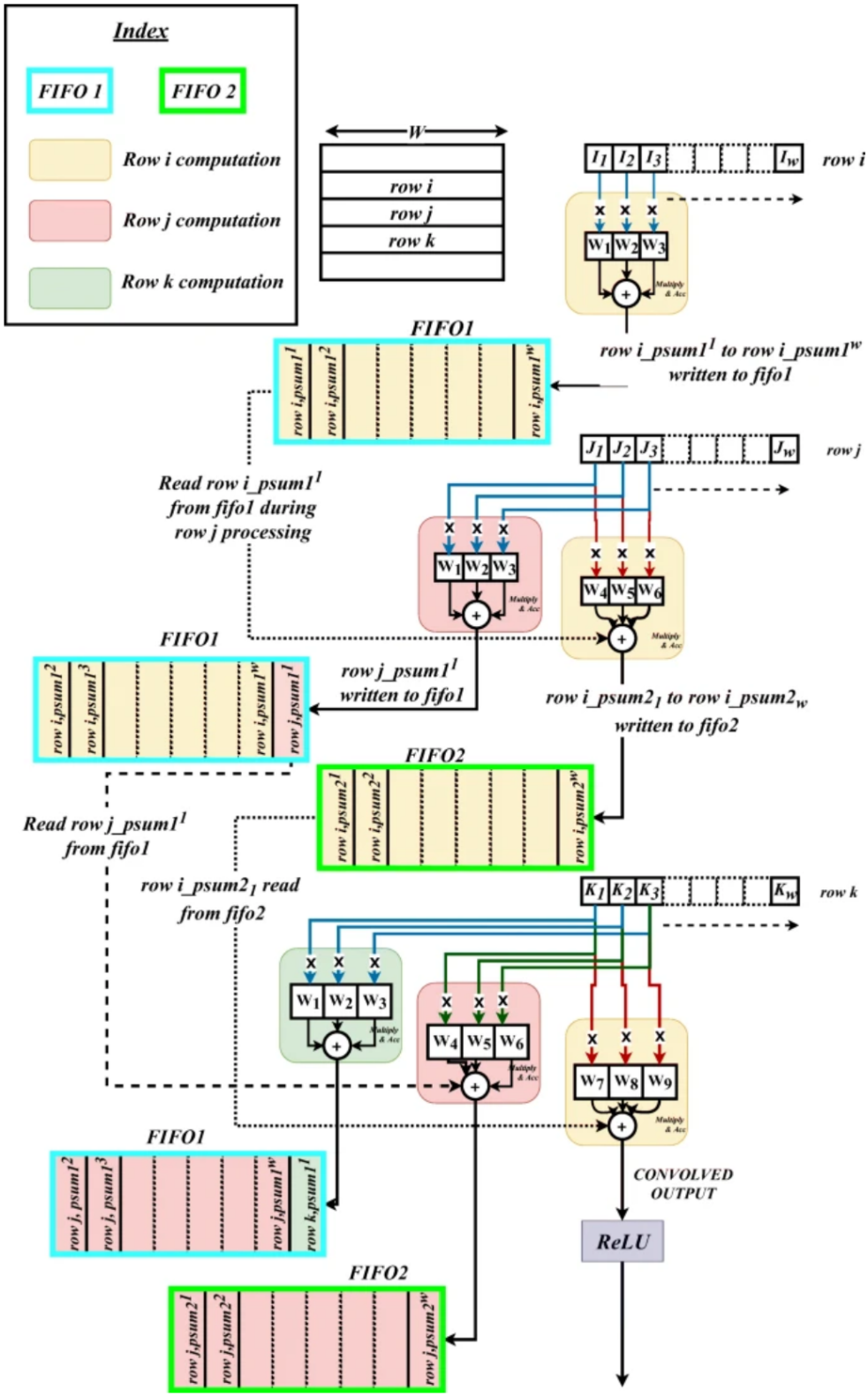

Figure 21: FIFO-psum architecture of convolution operation [9].

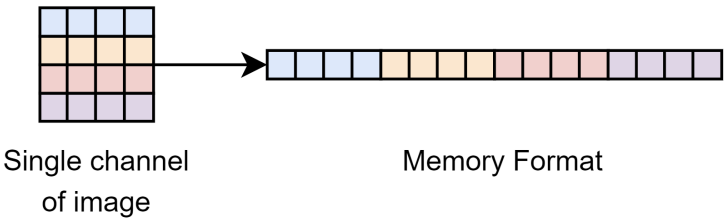

Figure 22: Memory format of input and output channels.

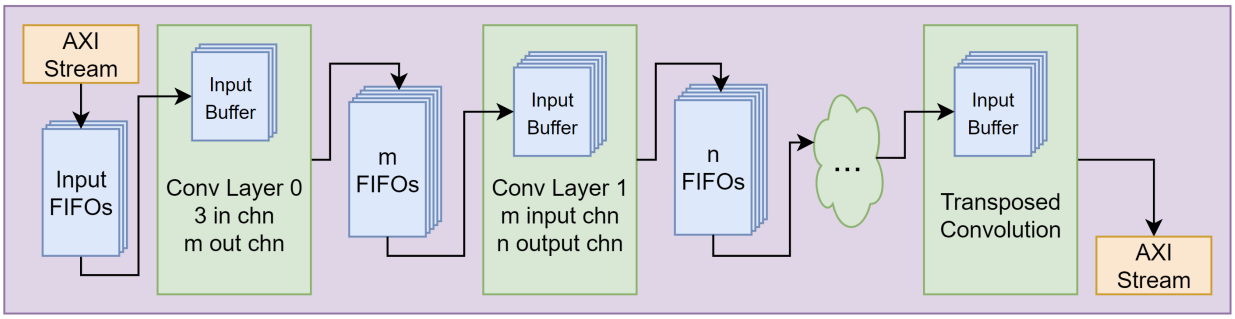

Figure 23: CNN architecture in Vitis HLS.

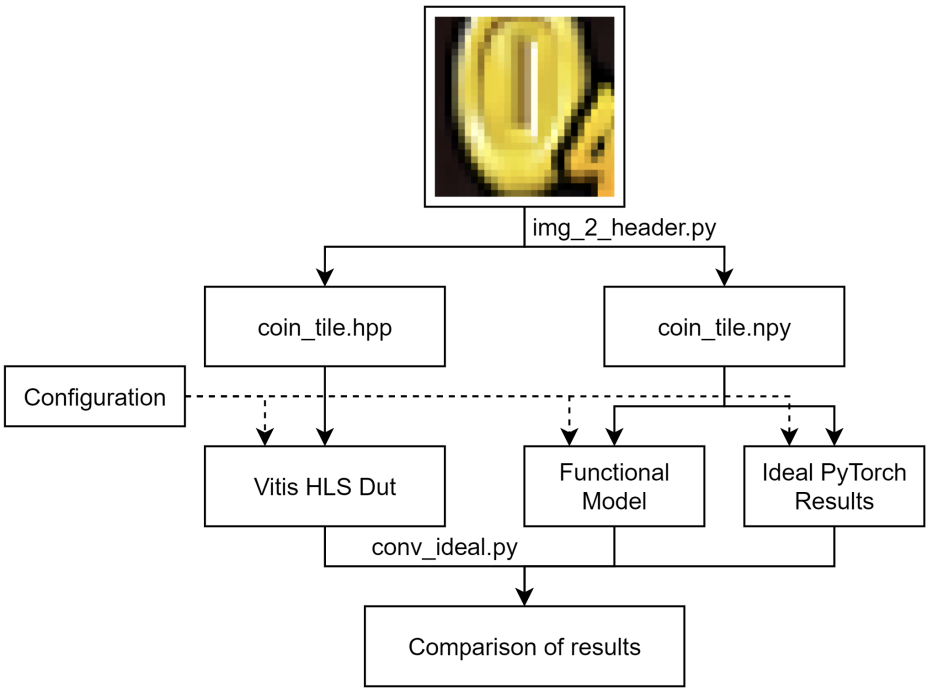

Figure 24: Diagram of testing setup for convolution.

first computes all simulation results using 32-bit floating-point numbers before transitioning to fixed-point computation.

Once the single convolution layer passed functional checks in Vitis HLS, we automated the process of layer development in Python. Given the number of input channels, output channels, kernel size, and input dimensions, a parameterized function generates synthesizable Vitis HLS C++ code. We then built up and

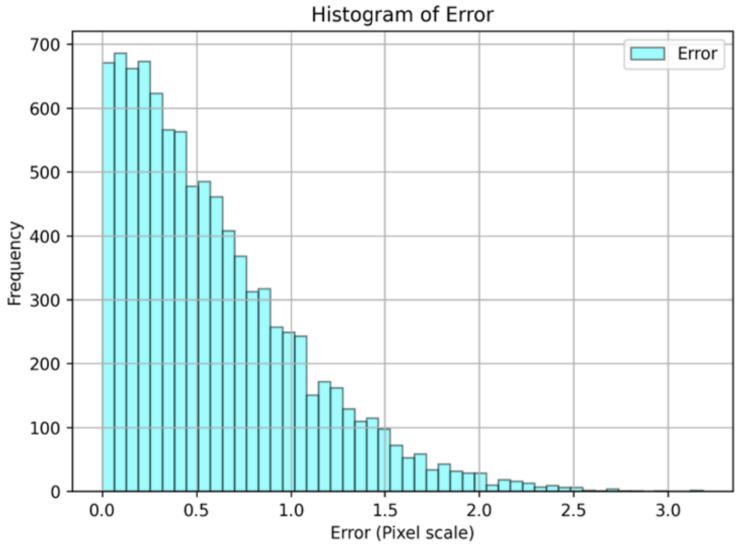

Figure 25: Accumulated DUT error for FSRCNN in Vitis HLS vs. PyTorch.

verified each layer of the network one at a time.

Once the team had written and implemented all convolutional layers, we began developing the final transposed convolution layer. Since transposed convolution can be emulated as normal convolution [42], the same FIFO-PSUM architecture performs the computation. The only difference is that the input tile requires the insertion of columns and rows of zeros between the actual data, and the weights from a trained deconvolution layer need to be flipped and transposed.

One key architectural issue that the team does not have time to fix is how we store weights on-chip. Currently, we declare all weights at compile-time and auto-instantiate them as LUT-ROM. This makes it easy to index any given weight, but it ultimately wastes LUT resources that could be used for implementing other functions. Additionally, all weight arrays are four-dimensional, which makes partitioning difficult. The solution to this issue is to flatten all weight arrays to one or two dimensions, store them in Block ROMs, and preload them into LUTs as needed for convolution computation.

Because the weights use a significant amount of LUTs, preliminary synthesis utilization reports show an overutilization of FPGA resources by 300%. Rather than redesigning the convolution architecture, we opt to shrink the neural network to match hardware constraints and, to the extent possible, image quality constraints.

The team uses the ap_fixed Vitis HLS library for all computations. Given that BRAM resources on the Zynq Ultrascale+ architecture have word lengths down to 18 bits, we use an 18-bit fixed-point data type to utilize the entire word. We manually quantize all network weights by simply casting to this fixed-point type, and the IP normalizes pixels coming into the network to [0,1), mimicking how PyTorch works. Rounding errors accumulate as the network progresses, so the team found it useful to create histograms to visualize how erroneous inferences are. Figure 25 illustrates the final result of implementing the simplified network.

3.4.3.3 Implementation Results

The team is still developing the IP that implements FSRCNN. Everything is functionally verified, but the team is currently debugging issues with running on hardware. Regarding resource utilization and estimated latency, a single 32 x 32 pixel tile requires approximately 516 µs to upscale. This means that in the 40 ms frame window required to achieve 25 FPS, only 10 tiles can be upscaled from FSRCNN.

3.5 Software Development

The application software portion of the MQP focuses on integrating all aspects of the project: configuring IP, capturing video frames, setting up video output, and dispatching tiles for upsampling. Since the project places a strong emphasis on running as quickly as possible, there is some deliberation about whether running bare metal or with PetaLinux is the right path. We decided to use PetaLinux, as it enables easy handling of I/O devices, allowing more time to be dedicated to application-specific tasks.

One of the main challenges of developing PetaLinux-based C++ applications is accessing hardware in the PL from userspace, specifically the AXI-DMA engine for sending pixel data and variance computation IP for programming the override controller. We thoroughly explore several options, including userspace input/output (UIO), developing custom loadable kernel modules, using existing drivers, and raw access via the /dev/mem device. Given a limited level of kernel programming experience and convoluted documentation on existing drivers, the team chooses to control hardware by accessing /dev/mem. This gives much finer-grained control over exactly what is happening. However, it also presents new challenges during debugging. The binaries must be run with sudo privileges, and writing to /dev/mem directly is generally discouraged in the Linux community due to the multitude of risks associated with it. It took a significant amount of debugging just to figure out that the programmable logic was not being programmed - writing to memory-mapped registers that didn't exist with /dev/mem would hang and crash the kernel, leaving minimal debug information and forcing a power cycle. That being said, it makes the transition into learning how to write PetaLinux applications easier, as it allows for bare-metal-style programming in an OS environment that can manage tedious I/O.

The general flow for controlling hardware involves creating a wrapper C++ class for each unit, within which memory access is contained and controlled through mmap calls to an open /dev/mem file descriptor. Each class then has interfaces such as start(), reset(), initialize(), etc., which internally handle reads and writes. Another important bug found during development was that each pointer to mmap'd hardware needed to be declared as volatile, otherwise, caching would cause significant headaches, especially when polling a register.

The main "glue logic" of the project lies in programming the AXI-DMA. Due to a lack of online resources, the team creates a wrapper class from scratch to control the engine and encapsulate low-level register programming. Since DMA operates with physical memory, the design needs to include a reserved buffer. The KV260 only has one DRAM chip, meaning the DMA has to read from the same memory that the OS has access to. Rather than requesting memory from the OS and creating a shared buffer, the team simply reserves 128MB of RAM in the device tree for general-purpose use with a fixed address range. During development, the AXI-DMA core was wired in a simple loopback configuration, allowing self-tests to verify functionality. The team first began developing the AXI-DMA C++ wrapper to target direct-register DMA (DR-DMA). In terms of the project, this meant that for every row in an image tile, a DMA transfer had to

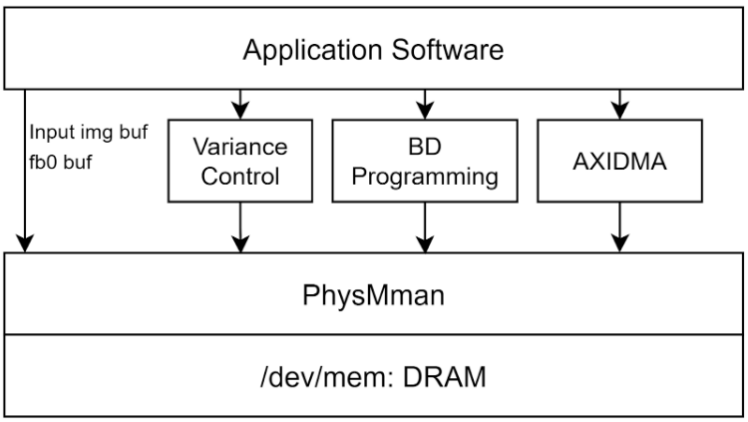

Figure 26: Application software high-level overview.

be manually programmed, which was tedious and slow. Once we had a good understanding of controlling DR mode, the team developed drivers for scatter-gather DMA (SG-DMA). SG-DMA allows for a transfer to be described via multiple buffer descriptors (BDs), each containing information on mini-transfers, including the source and destination addresses and buffer lengths. These BDs are then chained together in a linked list and individually programmed to describe a transfer. For example, sending a 32 x 32 pixel tile to the PL requires 32 buffer descriptors, each transferring 32 pixels. The DMA engine is given the address of the "start" and "tail" BDs - once the tail descriptor is set, it sequentially accesses BDs without CPU intervention.

However, this poses the need for even more userspace-controlled memory that can be shared with the DMA engine. To this end, the team develops a wrapper around the device-tree reserved memory: the PhysMman singleton (PMM). It first maps the reserved memory into userspace using mmap on a file descriptor to /dev/mem, then handles the allocation and freeing of fixed-size chunks for any process. This means that BDs are allocated and freed in a simple and modular fashion. A user can optionally supply a base address when requesting memory, in which case the PMM mmaps to that region of memory instead, adding another layer of memory encapsulation to all IP. Lastly, it provides easy management of buffers used in DMA transfers. Figure 26 describes the modular nature of the software.

3.6 Project Infrastructure

3.6.1 Build Server Setup and Repository Setup

For the actual implementation of the project, the team used a virtual machine (VM) running Ubuntu to act as a centralized build server. The VM is configured with Gnome desktop to provide a user interface for Vivado and Vitis HLS. One of the main challenges during setup was managing the various package dependencies between PetaLinux, Vivado, and Vitis HLS. Xilinx provides several user guides detailing how to manage these dependencies, but none of them work out of the box, and PetaLinux in particular requires significant manual intervention. Fortunately, this issue has been addressed by several online sources before. Learning to use the PetaLinux toolchain proved to be difficult, and even getting a simple image to boot on the target board was more challenging than originally anticipated. As such, a non-negligible amount of time was spent gaining experience with PetaLinux. UG1144 [43] and various online forum posts were helpful in resolving issues along the way [44].

The entire project is version-controlled using Git and is stored on GitHub. Whenever accidental pushes occur, i.e., those involving megabytes of build products, the team uses the git-filter-repo tool [45] to reduce the repository size. It is structured as seen in Figure 27. When development started, the repository was targeted at a single development board. However, the team had the foresight to see that cross-compatibility would be important, which undoubtedly saved countless hours after porting to the ZCU102. Thus, the project repository contains two subdirectories, one within each of the Vivado and PetaLinux parent directories, for each board.

3.6.2 Scripting and Cross-Platform Design

Another important aspect of the project's design is to automate as much of the process as possible using scripts. This is of paramount importance to allow the team to pick up work without interruption. Additionally, it provides a quick way to verify that changes in one area of the repository would not break anything else before pushing or merging new features to the main branch. Lastly, since Xilinx tools are known to be very powerful but also prone to bugs at times, treating all project directories as build products allows for quick regeneration. The team investigated creating a Jenkins server to monitor the repository, but other tasks, such as IP development, took precedence. The root of the repository has two main build scripts, one targeted for each development board. These scripts sequentially create all HLS projects and export the IP, create, populate, and export Vivado projects, configure the PetaLinux project with a generated hardware platform, and rebuild the entire PetaLinux image. These main scripts execute other bash scripts located within each subdirectory, keeping everything modular and organized.

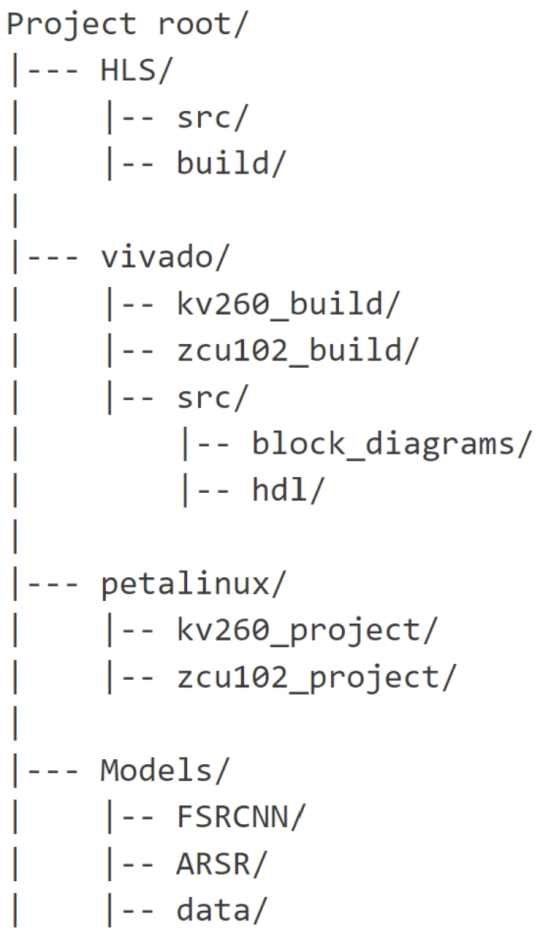

Figure 27: A diagram of the repository's structure.

The build script in the HLS directory retrieves information for each IP project from a JSON file, which contains the names of all source and testbench files, among other details. It then writes temporary project-creation and project-building Tcl scripts, which it passes to Vitis HLS on the command line. The script is cross-platform, since it is often easier to develop locally on a Windows machine.

The Vivado directory follows the same methodology of treating the entire project directory as a build product. Sources are stored externally and imported into each project during creation. This methodology is adapted from an article on the FPGA Developer's blog [46]. It is also cross-platform for the same reasons as Vitis HLS. Lastly, the PetaLinux build script takes the current project configurations and packages an image into a .wic format using a custom kickstart file (.wks), which can be flashed with a Windows machine.

The source code for the whole project is stored on GitHub at https://github.com/buchtawill/SR_MQP

4 HLS Verification Methods

4.1 Verification Methodology Overview

In addition to developing the IP blocks, we also established a methodology for the verification process, which we created in parallel with HLS development. At the project's start, we created the architectural specification shown in Figure 1 for use by both the design and verification teams. Afterward, the verification and design work proceed independently to avoid cross-referencing assumptions and to allow bugs to surface naturally. We built each testbench in two incremental steps. First, a self-test phase validates the testbench by running the ideal reference through the comprehensive C++ testbench predictor model. Any mismatch at this stage indicates an error in the driver, monitor, scoreboard, or predictor, not in the DUT. Once the testbench proves self-consistent, it can be used to test the synthesizable DUT.

We generate stimuli inside the testbenches using the C++ random distribution function. All data, including pixel intensities (ranging from 0 to 255), fractional values between 0 and 1 for the convolution feature extraction layer, and values from -1 to 1 for the remaining layers, follow a uniform distribution within their respective numeric ranges. Because stimulus parameters, such as image size, kernel size, stride, padding, and stream width, are configurable at the top of main(), we can recompile the same source code for different tile sizes without editing the DUT or verification infrastructure.

For each tile size, we execute a large Monte Carlo sweep: ten thousand iterations for bilinear interpolation and color space conversion, one thousand iterations per convolution layer and configuration, and one thousand end-to-end passes through the full CNN [37]. During each iteration, the driver packs stimulus into AXI-Stream transactions for color space conversion, interpolation, and full CNN, or FIFO transactions for individual convolution layers. It then invokes the DUT and unpacks the responses with the monitor. A Python or C++ predictor model produces an expected output vector, and the scoreboard compares it against the DUT output, recording the absolute value of the 8-bit error and classifying the trial as pass or fail under a user-selectable tolerance. Figure 28 illustrates the framework used for designing all the testbenches.

The entire verification process runs entirely automatically: the testbench generates multiple random test executions and sends them to both the DUT and the predictor. The predictor produces the reference output, and the scoreboard compares it with the DUT outputs before collecting final pass/fail tallies. When

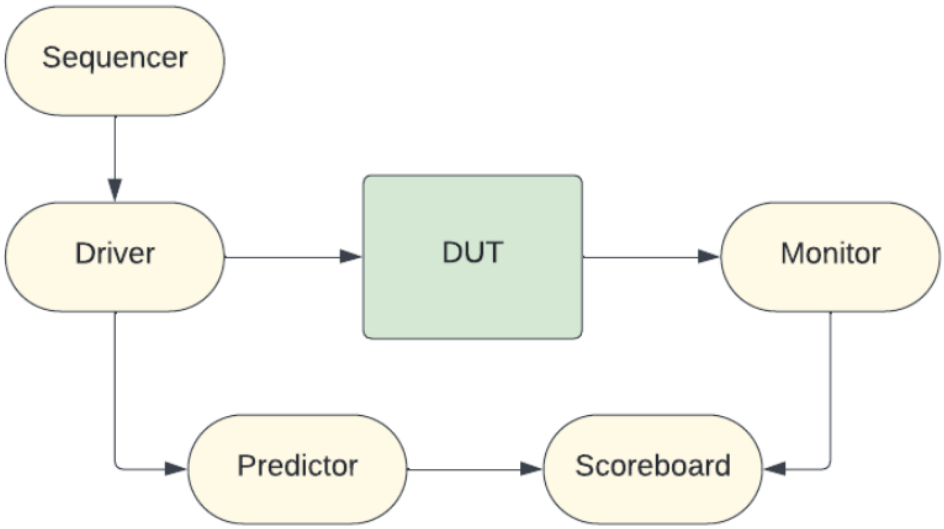

Figure 28: A block diagram showing testbench component structure.

discrepancies emerge, the testbenches print per-index deltas and the first few words of the stimulus, enabling rapid root-cause analysis. The team applies fixes to either the HLS or the testbench, then reruns the same Monte Carlo suite until statistical confidence returns to the desired threshold. This independent, self-checking, randomized, and highly parameterized methodology delivers deep functional coverage with minimal manual intervention. At the same time, the dual-resolution rerun proves that both the benches and the RTL scale correctly with image size.

4.2 Verification Testbenches

In each section of the system-wide block diagram, the team created a series of verification testbenches. The three verification environments behave like compact UVM-style self-checking systems, written entirely in standard C++ with some Python reference models. Every testbench shares the same logical flow, although each uses a specialized predictor.

First, a stimulus generator creates random vectors and scales them to the current runtime parameters, such as tile size, channel count, kernel size, stride, padding, and upscale factor. Next, a driver function packs this stimulus into the exact interface format expected by the DUT. For AXI-Stream designs, the driver assembles 8-, 32-, or 128-bit words, inserts back-pressure with in_stream.full(), and marks the final beat with TLAST. In a FIFO-based testbench, the driver pushes fixed-point values into an array of hls::stream objects, one per channel. Once the stream is primed, the HLS top function runs exactly once, mirroring the way Vitis HLS automatically schedules hardware. A monitor then reverses the packing logic to rebuild flat byte or fixed-point vectors, and the scoreboard compares the DUT output against a reference produced either by an internal C++ function or by an external PyTorch predictor. Across iterations, the scoreboard records pass and fail counts while printing helpful deltas on any mismatch.

Because every parameter that influences data shape or precision is configurable, it is exposed at the top of main(), and a single recompile exercises a new verification scenario. All the testbenches feature a golden pattern standard self-test, such as the coin-tile images, to ensure testbench functional accuracy before running simulation testing.

4.2.1 Color Space Conversion Testbench

The color space conversion testbench streams two-pixel macro-pixels through a 128-bit AXI interface. Its C++ predictor converts YUV samples into RGB triples and clamps values. The driver packs 16-byte bursts (including pad bytes) with proper TLAST, and the monitor discards pads and extracts valid RGB triples so the scoreboard compares exactly the three color bytes per pixel.

4.2.2 Bilinear Interpolation Testbench

The bilinear interpolation testbench illustrates this structure in three incarnations that differ only in data width. An 8-bit version streams one byte per cycle, a 32-bit version packs one padded pixel per beat, and a 128-bit version interleaves four padded RGB pixels into each word. All three benches reuse the same software predictor implementation and reproduce the HLS's exact byte ordering and handshake timing. A self-test with the coin-tile confirms bug-free compilation before the testbench fires random stimulus tiles through the pipeline.

4.2.3 CNN Testbench

The convolution verification testbench operates in two modes. In FIFO mode, each of the n input channels connects to the DUT through a dedicated FIFO stream. The testbench writes fixed-point values, then calls a PyTorch predictor in "FIFO" mode so the reference consumes the same floating-point activations that the DUT uses. By swapping the DUT "layer_name" string parameter, the same bench checks the feature-extraction, shrink, map, expand, or deconvolution layer. The full-network bench runs in AXI mode. It packs

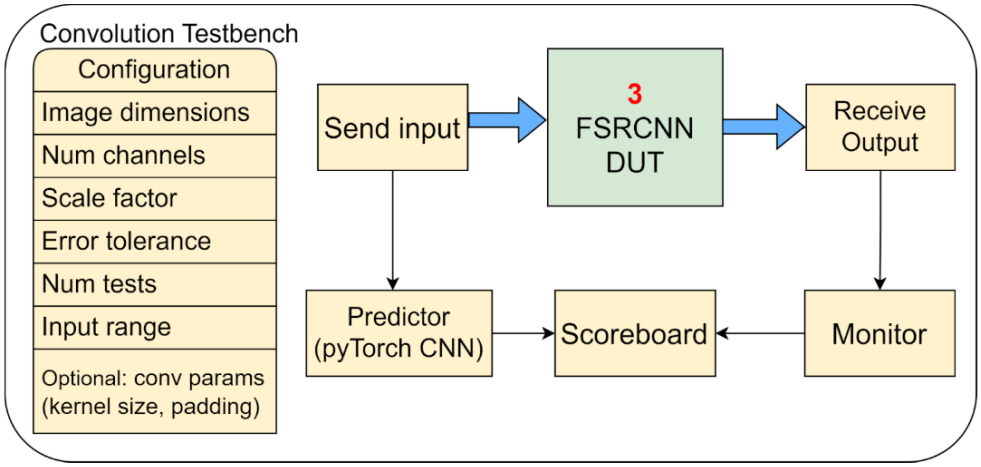

Figure 29: A high-level block diagram of the convolution testbench.

four RGB pixels into each 128-bit word, streams an entire tile into conv2d_top, and compares the HLS output, converted back to float, against a predicted PyTorch model reference that runs the complete FSRCNN with the identical weights. Because both hardware and software apply identical scaling and clipping rules, only a $\pm 3/255$ tolerance absorbs fixed-point rounding. Figure 29 below shows a high-level block diagram of the convolution testbench.

4.3 Maintainability

Several implementation techniques guarantee reuse and maintainability. Header-only utilities keep drivers and monitors identical across testbenches, isolating width-specific mechanics behind small helper functions. The driver and monitor abstractions for AXI and FIFO interfaces provide standardized test behavior, thereby boosting maintainability. Stimulus generation, reference execution, and checking operate purely in a data-driven manner, so updating resolutions, kernel sizes, or strides only requires editing constants in the main function. All testbench code along the DUT path remains HLS-compliant, enabling the same sources to support verification, C++ simulation, and cosimulation runs. These choices yield a compact yet powerful framework that tests HLS code with realistic workloads, offers high observability when mismatches arise, and scales effortlessly as designs evolve.

5 Results

Once each element of the system architecture was designed, implemented, and verified, the team analyzed both individual and system-wide results. Evaluating the results of the project's individual components was essential for understanding areas for improvement in future work, as well as any potential bottlenecks that may arise when the system is combined as a whole. The system-wide results offer a comprehensive view of the work accomplished and the system's overall effectiveness.

5.1 System Overview

The team currently has a demonstration able to run bilinear interpolation at 9.7 FPS on the KV260 development board. The physical interface for video input on the ZCU102 is still in development, so the demo on that board currently only upscales a single image from a file. The convolution IP uses more resources than what is available on the KV260, so it will only synthesize for the ZCU102. The IP is functionally verified in both Python and Vitis HLS; however, the team is currently debugging issues that arose when running cosimulation and testing on hardware. The resource utilization of all IP blocks at their current state is summarized in Table 5.

Table 5: Vitis HLS synthesis results compared to total resources on the ZCU102.

HLS IP Block	**BRAM**	**DSP**	**FF**	**LUT**
Variance/Color Conversion	11 (0.6%)	16 (0.6%)	955 (0.2%)	7018 (2.6%)
Bilinear Interpolation	89 (4.9%)	40 (1.6%)	8882 (1.6%)	17582 (6.4%)
Convolution	562 (30.8%)	811 (32.2%)	87488 (16.0%)	129034 (47.1%)
Total Used	**662 (36.3%)**	**867 (34.4%)**	**97325 (17.8%)**	**153634 (56.1%)**
Total Available	1824	2520	548160	274080

A significant bottleneck in our system was not fully understood until late in the project: transferring data to and from the programmable logic. This is because the DMA engine needs to fetch buffer descriptors every four stream beats. The latency to send a single 32 x 32 pixel tile into the PL is roughly 1,700 clock cycles, as measured with an ILA, or approximately 17 µs at 100 MHz. Receiving a tile takes approximately 3,600 clock cycles, or 36 µs at 100 MHz. With 368 tiles, the total latency, solely due to data transfer, is 19,500 µs, or nearly 20 ms. At 25 FPS, this is half of the total budget to achieve real-time processing. Additionally, it takes 54 clock cycles to transfer one row of a tile, whereas it only takes 4 clock cycles to send the data. This means that 93% of the transfer time is spent fetching buffer descriptors from DRAM.

5.2 Verification Results

The team verified each IP twice: first on 28 x 28 images, then again on 32 x 32 images. In each test, the testbenches generate thousands of fully random stimulus tiles, push them through the hardware, and compare the results against independent predictor reference models in the testbench while tracking the absolute difference on an 8-bit (0-255) scale.

For the bilinear interpolation module, three separate interface widths are used in succession: an 8-bit stream, a 32-bit padded-pixel stream, and a 128-bit packed stream, and ten thousand iterations are run for each width. With a stringent ±4 tolerance, 95 percent of all cases match; the remaining five percent show at most a single-digit error caused by fixed-point rounding in the interpolation calculations. When the tolerance was expanded by one to ±5, every remaining miscompare was eliminated, achieving a 100 percent pass rate and confirming that the discrepancies are numerical rather than functional. Importantly, the same success distribution is observed when the image dimension is increased to 32 x 32, indicating that the algorithm's error characteristics remain tile size-independent and that the wider data paths introduce no additional inaccuracy.

The color space converter delivers a flawless record. 10,000 randomized tiles are streamed, each containing YUV values, through the 128-bit driver interface onto the DUT. Every output triple matches the C++ predictor reference with a tolerance of ±1 for either image size. This result verifies that the block's implementation aligns with the architectural design.

Convolution verification was conducted in two stages. First, each layer of the CNN is isolated and tested individually in FIFO mode. One thousand trials per layer are run across two hand-picked configuration sets, covering kernel sizes, strides, and paddings, which are parameters likely to stress the datapath. All ordinary convolution layers agree with the floating-point PyTorch predictor within ±2, yielding a perfect 100-percent success rate. The deconvolution, or transpose-convolution, layer initially passes only 67 percent of trials at that ±2 threshold. By analyzing the difference between the DUT output and the predictor output, the team confirms a fractional mismatch caused by the precision bits of the fixed-point data. When the team raises the tolerance to ±3, it absorbs the small differences and restores a 100

In the second stage, the complete CNN is tested in AXI mode. One thousand random RGB images are streamed through the whole network, using both 28 x 28 and 32 x 32 pixel tiles, and the results are compared with the PyTorch predictor software model. With the tolerance set to ±3, the value justified by the deconvolution analysis, every single test matches, delivering a 100 percent success rate for end-to-end inference verification. Refer to Table 6 for a summary of all verification simulation runs on the different blocks and configurations of the testbenches.

Table 6: Simulation report for various verification testbenches.

Number of Tests	**IP Block (DUT)**	**Error Margin +/- (#/255)**	**Tests Passed**	**Tests Failed**	**Pass Rate**
10000	Bilinear Interpolation	1	0	10000	0%
		2	0	10000	0%
		3	7488	2512	74.88%
		4	9598	402	95.98%
		5	10000	0	100%
10000	Color Space Conversion	0	0	10000	0%
		1	10000	0	100%
1000	Convolution Layer: Feature Extraction	1	0	1000	0%
		2	1000	0	100%
1000	Convolution Layer: Shrink	1	0	1000	0%
		2	1000	0	100%
1000	Convolution Layer: Map 0,2,4	1	0	1000	0%
		2	1000	0	100%
1000	Convolution Layer: Expand	1	0	1000	0%
		2	1000	0	100%
1000	Convolution Layer: Transpose	1	0	1000	0%
		2	167	833	16.7%
		3	1000	0	100%
1000	Full CNN	1	0	1000	0%
		2	142	858	14.2%
		3	1000	0	100%

The statistical depth of the verification is worth noting. A Monte Carlo binomial estimate indicates that, with 10,000 samples, any hidden defect that manifests in more than 0.03% of the live data is almost certainly revealed [37]. This result demonstrates the robustness of the IP blocks' design and confirms their readiness for hardware implementation. Meanwhile, the verification testbenches identify several issues, including inconsistent AXI sideband usage between blocks, an overflow that wraps maximum intensities to zero, an off-by-one error in calculating the deconvolution output dimension, and an input length miscount in the bilinear interpolation. The testbenches report each defect to the IP block designer, who corrects them, as reflected in the improved pass rates in testbench reports.

These results provide the design team with strong quantitative evidence that the image-processing pipeline meets its numerical-precision targets and that its streaming interfaces behave consistently under random inputs. Moreover, the identical outcome across both tested resolutions confirms the testbenches'

configurability and the design's scalability. Remaining numerical deviations in the bilinear interpolation and some convolutional layers, such as the transpose-convolution, are fully explained by fixed-point rounding and lie well below one percent of full scale, an error budget deemed acceptable for the intended visual-quality requirements.

5.3 Example Upscaled Image

The final FSRCNN model successfully achieved 2x upscaling. An example output is provided in Figure 30. The final CUT output with an uncurated dataset shows promise for future implementations to successfully transfer the style from the Wii domain to the emulated domain. Figure 31 shows promising results when looking at Yoshi and the ground. These two parts of the frame appear to have better texture quality than the raw Wii frame.

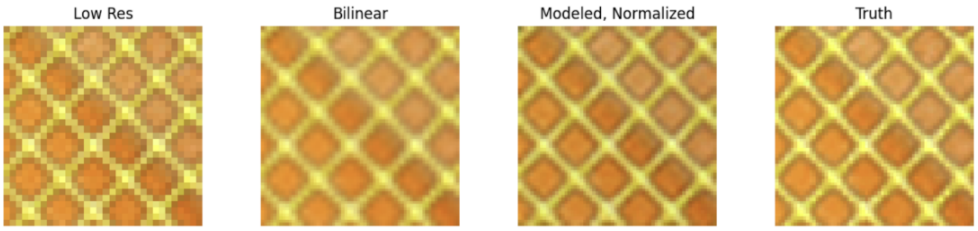

Figure 30: The output image that was generated after a training epoch, showing the final FSRCNN model upscaling a cropped emulated tile.

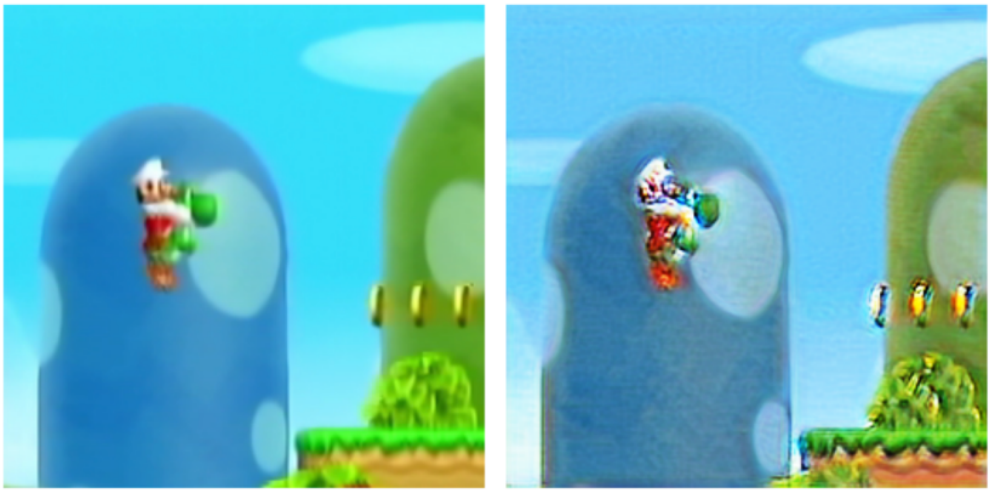

Figure 31: The left image shows a raw upscaled 2x Wii frame. The right image shows a style-shifted Wii frame using CUT

6 Discussion

6.1 Consideration of Public Health and Other Factors

6.1.1 Public Health

N/A

6.1.2 Public Safety

N/A

6.1.3 Public Welfare

N/A

6.1.4 Global Considerations

This design is meant to be adaptable for multiple platforms and use cases, but accessibility to resources such as FPGAs may be limited in certain regions, making it inaccessible as a product in areas where these resources are not as prominent.

6.1.5 Cultural Considerations

In light of the modification of the original content, there are ethical considerations regarding preserving the media created by Nintendo. Using artificial intelligence to alter video data, while in this case, was only done for research and personal use, would need to be further researched regarding patents and legal rights if a product like this system were to be sold commercially.

6.1.6 Social Considerations

This MQP modifies the original art from the artists and the studio. In the social context, this could be an issue for users who want to enjoy the original media. This could pose a social problem, as some people find the appeal of retro gaming appealing. Again, if this design were to be used commercially, a disclaimer would likely have to be present to inform users of the modifications being made to the media.

6.1.7 Economic Considerations

Regardless of how someone gets access to the game, Wii games are no longer sold by Nintendo, so there are no ethical qualms regarding taking money away from the original creator. On top of this, the hardware used to achieve this design

6.2 Recognizing Ethical and Professional Responsibilities

The primary ethical responsibility of the project is related to the resource utilization of GPUs and their role in the climate crisis. Large server racks, such as the Turing Cluster at WPI, generate a significant amount of heat that requires cooling. Server cooling is typically achieved using liquid cooling, which not only requires high water usage but also high energy consumption. This drives water scarcity, and high electricity usage can threaten nearby communities or raise utility prices [47].

6.2.1 Consideration of the Impact of Engineering Solutions of this MQP in Global, Economic, Environmental, and Social Contexts

Using standardized tools and protocols such as Vivado, Vitis HLS, PetaLinux, and AXI, promote reusable and portable code that can be used modularly on many platforms. However, using high-end FPGAs, like the ones detailed in this design, makes the system less accessible for personal use. However, avoiding the use of GPUs, instead opting for FPGAs, produces fewer economic concerns regarding resources and running costs. While not necessarily being a large consideration in the design, some parts of the system inherently address energy consumption; using two different upscaling algorithms, one fast and one slow, promotes energy efficiency, preserving resources and only using them when required. Contrasting concerns about changing source material, a design such as this one encourages preservation and popularization of older media. Some people's concerns with retro gaming involve their lower quality, making them less pleasing to play; increasing the resolution and quality, while retaining the original speed of the game, would likely bring a larger crowd back to older video games and revamp their life cycle.

6.3 Appropriate Incorporation of Engineering Standards

In the creation of this design, various engineering standards were utilized and manipulated for our use case. Multiple data type standards, such as IEEE floating-point, as well as arbitrary-precision fixed-point standards, were used in custom IP, verification, and model training. Video signal property standards, such as 576i, YUV 4:2:2, RGB 8:8:8, and RGB 5:6:5 were used to store and manipulate video throughout the pipeline. Lastly, AXI was used as the data transfer method between the programmable logic's hardware blocks. This made sending data much clearer and standardized compared to using a custom protocol, which could result in many more errors.

7 Future Work

Many areas of this project could be improved, as this year the team was starting from scratch. While there were efforts to improve efficiency and quality of results throughout the whole design, there is always room for improvement. For instance, the hardware implementation of the FSRCNN is not yet fully complete, and the evaluation board that helps decode and split the signal has not been implemented either. Significant work can be done to expand style transfer for training, enhance both methods of upscaling, and refine the system architecture.

As shown in the results, the output of an upscaled crop using FSRCNN provided accurate upscaling but did not provide a higher sharpness for textures. This problem was attempted to be solved by using the CUT training network. This approach did not work for our use case, as the Wii frame and emulated frame pairs were not curated and lacked similar enough attributes to have their style transferred. In the future, using preprocessed data that is curated for the pairs could provide better results and improve the texture quality and sharpness of the game when upscaling using FSRCNN.

Regarding hardware, several steps can be taken to enhance the efficiency of the bilinear interpolation module. Analyzing the waveforms reveals that most of the process time is spent on calculations and transferring pixels out, rather than reading pixels in. One way to decrease this time would be to start pixel transfer out before calculations have completed. Given that calculations already happen using subsets of the image, the top rows of the tile are fully calculated before calculations start on the bottom sections. As such, those fully interpolated sections could begin to transfer out earlier. While this wouldn't decrease the time required for either the calculations or the transfer, overlapping the two steps would lead to an overall decrease in process time. Additionally, more time should be spent reducing the resource utilization of the calculations. Decreasing usage either through refactoring how calculations are performed or the general structure of the block would allow for more pipelining, as well as more resources that could be allocated to the CNN for tile processing. This would allow for an overall speed up both within the bilinear interpolation block and within the CNN. While it may not seem important to increase the speed of the bilinear interpolation block if it is already calculating at real-time, it is necessary if future iterations want to increase the frame rate of the output video from the original video, or if the desired output video resolution increases.

The primary area for improving the FSRCNN IP block is to manage the implementation of weights. Currently, all weights are defined at compile time, meaning that loading new weights would require re-synthesizing and re-implementing the FPGA bitstream. Additionally, since the weights are declared as four-dimensional arrays, Vitis HLS automatically infers that they should be implemented as LUT-ROMs, wasting resources. A simple fix would be to flatten each weight array for easy storage in BRAM and create a low-level cache of immediate weights used in computation. Lastly, since the IP architecture is inherently pipelined per row of a tile, upscaling larger-sized tiles would reduce latencies created by restarting the pipeline.

As mentioned in the results section, the main bottleneck in the team's design is the DMA transfer to and from the programmable logic. The high bandwidth associated with 128-bit stream data width means that DMA buffer descriptors need to be fetched after just 4 beats, essentially killing any gain from a wide data width. Now that the team has a ZCU102 development board, there is an abundance of BRAM resources that can be utilized to develop a data reformatter to send tiles in hardware, rather than doing so in software. This would significantly reduce the latency introduced by the need to fetch buffer descriptors for every 4 stream beats of data. Lastly, with increased BRAM resources, it is more feasible to upscale larger tiles, i.e., 64 x 64 to 128 x 128, reducing the impact of transfer latency. Another useful improvement to the system design would be to draw frames to the screen via the on-chip Mali GPU, reducing buffering delays associated with the frame buffer device and allowing the output to use the RGB 8:8:8 color space, increasing color variety and quality.

One final improvement is to provide additional hardware to support the FPGA board. The Analog Devices EVAL-ADV7182AEBZ was chosen as the video decoder and splitter for the system. The intention with this board was to obtain the composite video from the Wii, decode it into digital signals, and split the output to a YPbPr-to-HDMI converter and the pin headers. The YPbPr output would be sent to one

monitor, displaying the native resolution of the Wii, while the pin headers transmit video data and timing signals to the ZCU102. The 8-bit data signals are accompanied by VSYNC, HSYNC, and a Line-Locked Clock (LLC), all of which are necessary signals for transmitting video [48]. Prebuilt IP blocks in Vivado, specifically the Video Timing Controller and Video In to AXI4-Stream, can be used together to stabilize video timing and convert the raw video into AXI-Stream format for transmission into the system. The team began working with this evaluation board but ran out of time to get it working before the project's completion. Comparing this to the current software implementation of video input reading, the evaluation board would help to decrease the video latency introduced by V4L2. This is because the API waits for a frame-finished signal before starting to stream data. Incorporating this evaluation board into the design would enable the constant streaming of pixels into the pipeline, limited only by the decoder's read speed. Rather than waiting for a full frame, the tiler can begin sending tiles out to the hardware pipeline as soon as it receives 32 rows of pixels.

8 Conclusion

This project focused on identifying, implementing, and analyzing upscaling techniques when applied to a *Super Mario Bros. Wii* video output from a Wii. While image upscaling is not a new challenge in the image processing space, it remains an issue that has yet to be efficiently solved for old, low-resolution media, especially from video games, which exhibit a different level of image variance and complexity compared to filmed media. The complexity of this problem necessitated a novel approach, and the development of a system using HLS to be implemented on an FPGA provided the necessary versatility and rapid development time to address these challenges.

To begin the project, the team researched previous implementations of image processing and upscaling techniques, narrowing the applicable methods to bilinear interpolation and FSRCNN. Research has shown the importance of utilizing multiple upscaling techniques to address memory and speed limitations.

Once the desired techniques were identified, the team developed a methodology to implement upscaling methods, along with the system components necessary to support them. This included splitting the image into tiles, converting the Wii video from a YUV to RGB color space, determining the variance of tiles depending on complexity, and developing a machine learning model that could generate the weights necessary for a CNN. Each of these processes was essential to the overall success of the project, as video upscaling would not have been possible without a robust system to integrate upon. Each of these components required testing to fully verify its results before being implemented on hardware.

When fully tested and implemented, the system yielded a 100% accuracy on both the bilinear interpolation and FSRCNN; however, resource utilization was still large, and computation time and board limitations meant the system only produced a 10 FPS output. Despite this, the project proved successful, as it demonstrated the potential of the architecture and accurately upsampled a video using an FPGA and machine learning to support a neural network. Future work should focus on enhancing the speed of bilinear interpolation when implemented in the ILA and reducing the resource utilization of the FSRCNN to enable its operation without compromising the model's quality.

Appendices

A List of Acronyms

Acronym	Definition
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUT	Device Under Test
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FSRCNN	Fast Super-Resolution Convolutional Neural Network
HDL	Hardware Description Language
HLS	High-Level Synthesis
HR	High Resolution
ILA	Integrated Logic Analyzer
IP	Intellectual Property
LUT	Look Up Table
LR	Low Resolution
ReLU	Rectified Linear Unit
RTL	Register Transfer Level
URAM	Ultra Random Access Memory
UVM	Universal Verification Methodology

References

[1] C. Dong, C. C. Loy, and X. Tang, "Accelerating the Super-Resolution Convolutional Neural Network," arXiv, Tech. Rep. arXiv:1608.00367, Aug. 2016, arXiv:1608.00367. [Online]. Available: http://arxiv.org/abs/1608.00367

[2] "What are RGB and YUV color spaces? I DEXON Blog," Apr. 2022. [Online]. Available: https://dexonsystems.com/blog/rgb-yuv-color-spaces

[3] N. Pai, "Understanding RGB, YCbCr and Lab Color Spaces," 2024. [Online]. Available: https://medium.com/@weichenpai/understanding-rgb-ycbcr-and-lab-color-spaces-f9c4a5fe485a

[4] TaranVH, "Nearest Neighbor Scaling," Jan. 2023. [Online]. Available: https://community.adobe.com/t5/premiere-pro-ideas/nearest-neighbor-scaling-sampling-simple-and-vital/idi-p/13515843

[5] "Zoom Images : Nearest Neighbour & Bilinear Interpolation," Jan. 2012. [Online]. Available: https://tjeyamy.blogspot.com/2012/01/zoom-images-nearest-neighbour-bilinear.html

[6] P. Ratan, "What is the Convolutional Neural Network Architecture?" Oct. 2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/10/what-is-the-convolutional-neural-network-architecture/

[7] Siemens, "Pipelining." [Online]. Available: https://hls.academy/topics/pipelining/hls.academy/topics/pipelining/

[8] A. Szczepanek, "Bilinear Interpolation Calculator," Jun. 2024. [Online]. Available: https://www.omnicalculator.com/math/bilinear-interpolation

[9] V. Panchbhaiyye and T. Ogunfunmi, "An Efficient FIFO Based Accelerator for Convolutional Neural Networks," *Journal of Signal Processing Systems*, vol. 93, no. 10, pp. 1117–1129, Oct. 2021. [Online]. Available: https://link.springer.com/10.1007/s11265-020-01632-0

[10] "Browser Display Statistics." [Online]. Available: https://www.w3schools.com/browsers/browsers_display.asp

[11] Plastics Industry Association and Consumer Technology Association, "4K Ultra HDTV household penetration in the United States from 2014 to 2018 [graph]," https://www.statista.com/statistics/736142/4k-ultra-hdtv-us-household-penetration, Jun. 2018, accessed: 2025-04-29.

[12] B. Choi, "Pixel Perfect: RTX Video Super Resolution Now Available for GeForce RTX 40 and 30 Series GPUs," Feb. 2023. [Online]. Available: https://blogs.nvidia.com/blog/rtx-video-super-resolution/

[13] Elvtr, "Top of the leaderboard: the most popular retro video games from the 1980s and 1990s." [Online]. Available: https://elvtr.com/blog/top-of-the-leaderboard-the-most-popular-retro-video-games-from-the-1980s-and-1990s

[14] C. Dong, C. C. Loy, K. He, and X. Tang, "Image Super-Resolution Using Deep Convolutional Networks," Jul. 2015, arXiv:1501.00092. [Online]. Available: http://arxiv.org/abs/1501.00092

[15] Z. He, H. Huang, M. Jiang, Y. Bai, and G. Luo, "FPGA-Based Real-Time Super-Resolution System for Ultra High Definition Videos," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO, USA: IEEE, Apr. 2018, pp. 181–188. [Online]. Available: https://ieeexplore.ieee.org/document/8457651/

[16] J. Van Ouwerkerk, "Image super-resolution survey," *Image and Vision Computing*, vol. 24, no. 10, pp. 1039–1052, Oct. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0262885606001089

[17] J. Naranjo-Torres, M. Mora, R. Hernández-García, R. J. Barrientos, C. Fredes, and A. Valenzuela, "A Review of Convolutional Neural Network Applied to Fruit Image Processing," *Applied Sciences*, vol. 10, no. 10, p. 3443, May 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/10/3443

[18] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A Convolutional Neural Network Cascade for Face Detection," in *Conference on Computer Vision and Pattern Recognition*, Boston, 2015, pp. 5325–5334.

[19] "Introduction to Color Spaces in Video | Matrox Video." [Online]. Available: https://video.matrox.com/en/media/guides-articles/introduction-color-spaces-video

[20] "What Is Variance in Statistics? Definition, Formula, and Example." [Online]. Available: https://www.investopedia.com/terms/v/variance.asp

[21] S. Fadnavis, "Image Interpolation Techniques in Digital Image Processing: An Overview," *International Journal of Engineering Research and Applications*, vol. 4', no. 10, pp. 70–73, Oct. 2014. [Online]. Available: https://www.researchgate.net/profile/Shreyas-Fadnavis/publication/301889708_Image_Interpolation_Techniques_in_Digital_Image_Processing_An_Overview/links/5abcee20a6fdcccda656f974/Image-Interpolation-Techniques-in-Digital-Image-Processing-An-Overview.pdf

[22] V. Patel and K. Mistree, "A Review on Different Image Interpolation Techniques for Image Enhancement," *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, no. 12, Dec. 2013. [Online]. Available: https://www.academia.edu/38061521/A_Review_on_Different_Image_Interpolation_Techniques_for_Image_Enhancement

[23] Amanrao, "Image Upscaling Using Bicubic Interpolation," Sep. 2023. [Online]. Available: https://medium.com/@amanrao032/image-upscaling-using-bicubic-interpolation-ddb37295df0

[24] A. Giachetti and N. Asuni, "Real-Time Artifact-Free Image Upscaling," *IEEE Transactions on Image Processing*, vol. 20, no. 10, pp. 2760–2768, Oct. 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5741850/

[25] S. Pendhari, "Connected Layer vs Fully Connected Layer," Dec. 2024. [Online]. Available: https://medium.com/@sarahpendhari/connected-layer-vs-fully-connected-layer-32b4cbb29824

[26] Y. Ma, N. Suda, Cao, Yu, J.-s. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7577356/

[27] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Accelerator," Dec. 2018, arXiv:1712.08934. [Online]. Available: http://arxiv.org/abs/1712.08934

[28] D. Unzueta, "Fully Connected Layer vs. Convolutional Layer: Explained." [Online]. Available: https://builtin.com/machine-learning/fully-connected-layer

[29] Machine Learning in Plain English, "Convolutional Neural Network — Lesson 9: Activation Functions in CNNs," Jun. 2023. [Online]. Available: https://medium.com/@nerdjock/convolutional-neural-network-lesson-9-activation-functions-in-cnns-57def9c6e759

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," Feb. 2015, arXiv:1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852

[31] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2015, pp. 161–170. [Online]. Available: https://dl.acm.org/doi/10.1145/2684746.2689060

[32] 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)*. New Cairo City, Egypt: IEEE, Dec. 2021, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/9664920/

[33] D. G. Bailey, "The advantages and limitations of high level synthesis for FPGA based image processing," in *Proceedings of the 9th International Conference on Distributed Smart Cameras*. Seville Spain: ACM, Sep. 2015, pp. 134–139. [Online]. Available: https://dl.acm.org/doi/10.1145/2789116.2789145

[34] AMD, "Vitis High-Level Synthesis User Guide." [Online]. Available: https://docs.amd.com/r/2024.2-English/ug1399-vitis-hls/syn.directive.unroll

[35] Intel, "Intel® High Level Synthesis Compiler Standard Edition: Reference Manual," Dec. 2019.

[36] Siemens, "UVM - Universal Verification Methodology." [Online]. Available: https://verificationacademy.com/topics/uvm-universal-verification-methodology/verificationacademy.com/topics/uvm-universal-verification-methodology/

[37] E. Aldridge, "Understanding the Monte Carlo Analysis in Project Management," Jun. 2023. [Online]. Available: https://projectmanagementacademy.net/resources/blog/understanding-the-monte-carlo-analysis-in-project-management/

[38] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks," 2017. [Online]. Available: https://arxiv.org/abs/1703.10593

[39] T. Park, A. A. Efros, R. Zhang, and J.-Y. Zhu, "Contrastive Learning for Unpaired Image-to-Image Translation," 2020. [Online]. Available: https://arxiv.org/abs/2007.15651

[40] "AMBA AXI-Stream," 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest/

[41] Y. Yang, P. Yuhua, and L. Zhaoguang, "A Fast Algorithm for YCbCr to RGB Conversion," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 4, pp. 1490–1493, Nov. 2007. [Online]. Available: https://ieeexplore.ieee.org/document/4429242/

[42] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016. [Online]. Available: https://arxiv.org/abs/1603.07285

[43] "AMD Technical Information Portal." [Online]. Available: https://docs.amd.com/r/2023.1-English/ug1144-petalinux-tools-reference-guide/Overview

[44] W. Knitter, "Vivado, Vitis, & PetaLinux 2023.1 Install on Ubuntu 22.04," Aug. 2023. [Online]. Available: https://www.hackster.io/whitney-knitter/vivado-vitis-petalinux-2023-1-install-on-ubuntu-22-04-ab28da

[45] E. Newren, "newren/git-filter-repo," Apr. 2025, original-date: 2018-08-21T15:40:09Z. [Online]. Available: https://github.com/newren/git-filter-repo

[46] J. Johnson, "Version control for Vivado projects," Aug. 2014. [Online]. Available: https://www.fpgadeveloper.com/2014/08/version-control-for-vivado-projects.html/

[47] A. Zewe, "Explained: Generative AI's environmental impact," Jan. 2025. [Online]. Available: https://news.mit.edu/2025/explained-generative-ai-environmental-impact-0117

[48] Analog Devices, *ADV7182A*, 2025, accessed: Apr. 22, 2025. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADV7182A.pdf