TRANSLATION OF CNN MODEL FOR HARDWARE ACCELERATION

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Nikhil Mhatre, Devin Singh, Junze Zhou

MEng Field Advisor: Hunter Adams

Degree Date: May 2024

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Translation of CNN Model for Hardware Acceleration

Author: Nikhil Mhatre, Devin Singh, Junze Zhou

Abstract:

Convolutional Neural Networks (CNN) help process raw information from the environment and transform it into actionable knowledge. Currently, general-purpose CPUs are commonly used as a software platform for running inference with CNNs. This is due to the simplicity of development and training with common C++ and Python programming languages. CNNs perform highly repetitive and computationally intensive calculations that specialized hardware can use for better performance. We believe a hardware description language (HDL) implementation of a CNN on a Field Programmable Gate Array (FPGA) can utilize the unique architecture for a faster prediction time while maintaining accurate results.

Executive Summary

Our team formed over a joint interest in learning how specialized hardware could be utilized to improve a computationally intensive task. With a shared background in FPGA hardware and an interest in learning more about CNNs, we believed we could combine the two and develop a real-time video processing application.

We began our project by exploring the features of the Zybo Z7 FPGA hardware and PCAM to get video data to display on a monitor. This required us to install the Xilinx Vivado IDE to develop and run Verilog to test the PCAM, FPGA board, and HDMI output connection. After numerous failed attempts to set up the PCAM, we transitioned to gathering video data by streaming it from our laptop camera and into the FPGA with the HDMI input. We achieved video pass-through using the Zybo Z7 HDMI ports. Additionally, we successfully compiled a Petalinux kernel on the FPGA's dual-core ARM Cortex-A9 processor and could boot the processor from a Linux image.

After understanding the capabilities of the Zybo Z7 FPGA, we solidified the group's direction to implement an object classification CNN. We started to develop our understanding of a CNN through the Yolov3 model for object detection and classification. Through online resources, we implemented the model with images and upgraded the application by using OpenCV to perform real-time detection through our webcam.

This subproject exposed the memory size requirements to host the CNN, resulting in a decision to switch to the Tiny Darknet model for image classification since it could fit on the FPGA. At this time, we also researched Vivado's High-Level Synthesis tool to convert the Tiny Darknet model written in C++ to Verilog. With memory, loop, and data communication optimizations in the C-Level code, we ensured that the translated program efficiently utilizes resources on the destination FPGA board. The last required upgrade in our project was to implement the network on the Zedboard FPGA since it contained more hardware resources

than the synthesizable RTL needed to run inference. Our work culminated after we generated a bitstream file that we used to program the Zedboard FPGA. We achieved a 25x speed up in prediction time and maintained over 50% accuracy when implementing the CNN algorithms on an FPGA and using HLS for hardware system design.

Individual Contributions

Nikhil contributed to setup of Xillinx Vivado and running demo programs on the Zybo Z7. Nikhil also implemented the Yolov3 convolutional neural network with OpenCV to get object detection and classification running. After understanding the network memory issues, Nikhil worked to implement running the Tiny Darknet network written in C++ on a CPU and using HLS for translation to Verilog. Finally, Nikhil worked with Devin to get statistics on the image classification performance speed up with different images.

Devin worked on the Petalinux compilation process and contributed to HDMI/PCAM demo debugging. After being provided with the completed files, Devin performed HLS conversion of the Darknet CNN into Verilog. After conversion, Devin worked closely with Nikhil to perform Zedboard setup, generate a bitstream file, and execute the completed demo on the Zedboard.

Junze helped develop the translation infrastructure, including data representation conversion from floating point to fixed point and data organization by packing four 8-bit data to reduce the hardware utilization of FPGA when HLS optimization directives greatly increase hardware resources usage.

Introduction

Basic structure and features of Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of deep neural networks that are especially effective for processing data that has a grid-like topology, such as images. CNNs are widely used in the field of computer vision, where they have been successful in tasks such as image recognition, object detection, and segmentation.

Typical CNNs contains several key components. Convolutional layers are the core blocks of CNNs, they apply several filters to detect different features, such as edges, textures, and color. Following convolution, pooling layers reduce the dimensionality of each feature map but retain the most important information. Max pooling, which selects the maximum value from a group of pixels, is the most common method. After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks. These layers are typically placed near the end of CNN architectures [9].

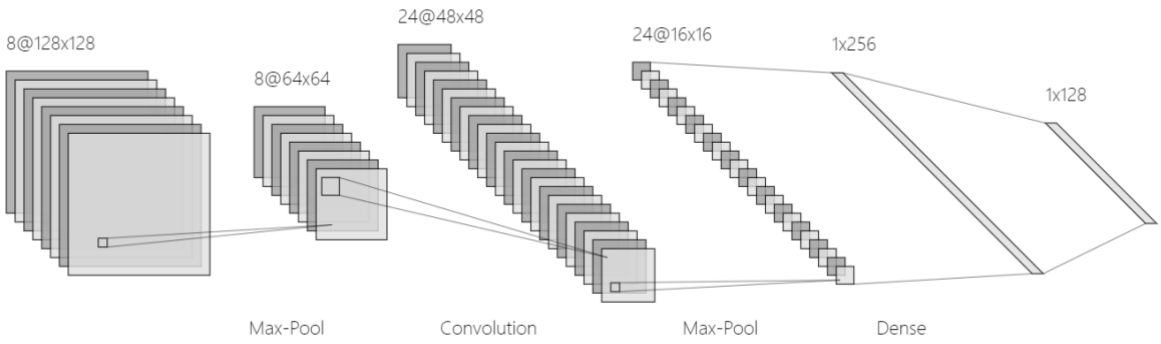

Figure 1. Components of CNNs [7]

All three typical components of CNNs are computationally intensive but also highly amenable to parallel processing. For convolutional layers, the same filter is applied independently to each part of the input image, and different filters can be applied in parallel,

this process is inherently parallelizable. Similarly, pooling operations over different parts of the input can be performed independently and in parallel, which significantly speeds up the processing. Fully Connected Layers mainly involve multiplications, which are also highly parallelizable.

CNNs Models and Applications

In the realm of image classification, several CNN models stand out for their distinct architectures. AlexNet pioneered the use of deep CNNs with consecutive convolutional layers, max-pooling, dropout, and fully connected layers [5]. ResNet introduces skip connections to facilitate the training of very deep networks [3]. SqueezeNet utilizes fire modules with 1x1 and 3x3 filters to significantly reduce model size while maintaining high accuracy [4]. These models form the backbone of current advances in image processing and are widely implemented in various frameworks.

Apart from image classification, CNNs are used in various applications due to their ability to automatically learn spatial hierarchies of features from data. In the field of object detection, there are many popular networks like YOLO, where a CNN is used to classify and locate multiple objects within an image. In the application of robot development, vision-based navigation employs CNNs to instruct the movement of robots. For medical applications, CNNs can also be used to detect conditions like pneumonia, tumors, and retinal diseases by identifying subtle patterns in medical images.

Software implementations of CNNs

Python and C++ are the most popular programming languages used in the development of CNNs. Python boasts powerful libraries for machine learning such as TensorFlow, PyTorch, and Keras. These libraries simplify the process of building, training,

and deploying CNNs with high-level abstractions. C++ is a lower-level language compared to Python, which allows for finer control over system resources and memory management. This can lead to performance improvements, which are critical in production environments, especially for real-time applications. For applications requiring CNN models to run on embedded devices or mobile platforms, C++ is often preferred due to its ability to produce compact and efficient executables that are better suited for limited-resource environments.

Hardware platform to implement CNNs

As mentioned above, a convolutional neural network (CNN) is a typical computationally intensive algorithm with potential high parallelism. Specifically, there are six levels of loops for convolutional operation as shown below, each of which can be unrolled or pipelined to improve performance to a great extent.

```
for(int o=0;o<out_channel;o++)
    for(int i=0;i<in_channel;i++)
        for(int x=0;x<out_feature_size;x++)
            for(int y=0;y< out_feature_size;y++)
                for(int u=0;u<kernel_size;u++)
                    for(int v=0;v< kernel_size;v++)
                        out[o][x][y] += in[i][x+u][y+v]*weight[u][v];
```

Figure 2. Multiple levels of loops in convolution with high parallelism

When exploring the high level of parallelism within CNNs, the architectural differences of CPUs, GPUs, and FPGAs can lead to significant variations in performance and efficiency. CPUs are less efficient for CNN tasks due to their sequential processing nature, resulting in higher power consumption and slower performance. GPUs excel in handling the intensive matrix and vector operations typical in CNNs, offering faster inferencing and better performance per watt, making them ideal for scenarios demanding high throughput. FPGAs provide configurable hardware that can be tailored specifically to the network architecture,

achieving high efficiency and low power consumption by maximizing operational parallelism. These specialized devices often outperform general-purpose GPUs in power efficiency, especially in customized applications, balancing the trade-off between flexibility, cost, and power usage. [10]

Potential Benefits of Hardware Implementation

A Field Programmable Gate Array (FPGA) is an array of reprogrammable logic blocks that are programmed at runtime to represent a certain logic design. FPGA's reconfigurability at the bit level allows for exact hardware, making it an ideal hardware solution for real-time video processing compared to CPU implementations. The ability to build a device specific for video processing requirements allows unnecessary hardware to be eliminated and distinct optimizations to be implemented.

One of the primary uses of FPGAs is for its hardware acceleration of high computation programs. This is because FPGAs allow for parallel processing and hardware specialization. Our project is a real-time video-processing accelerator that will be designed on the Zybo Z7 FPGA hardware. The hardware design of an accelerated Convolutional Neural Network (CNN) can be used to improve computational speed in the growing machine-learning industry. The inference procedure in a CNN has a high number of matrix multiplication that can be sped up with the FPGA's ability of parallel processing. Additionally, hardware specialization reduces the overhead of on-chip utilization by matching the needs of the CNN architecture [1]. With these FPGA features, we will be able to achieve a performance acceleration compared to a general-purpose CPU implementation.

Initial Design Problem

FPGAs for Real-Time Video Processing

CNNs can help with recognition, object detection, and classification when performing real-time video processing. CNNs can receive input frames from video, and in real-time perform identification and classification. Our team was initially interested in developing the infrastructure to support real-time video processing and use the FPGA to accelerate specific applications. CNNs are highly parallel structures, containing multiple layers with nodes that are performing calculations at the same time [9]. FPGA architecture can be exploited to use the inherent parallelism that exists within them. By parallelizing node calculations between layers, we believe we can deploy a CNN that performs real-time object detection on a FPGA faster than a typical CPU.

Zybo Z7 FPGA Components

We started with the Zybo Z7 FPGA hardware to learn about the setup and SoC architecture of an FPGA development board. One of the components on the SoC of the Zybo Z7 is a dual core ARM Cortex A9 processor. The ARM processor will be used for interfacing with our programmable logic and for preprocessing video input. The height and width of the images sent to the network must be reduced before classification can begin. Preprocessing is intended to be completed using resizing methods, such as binary interpolation. Additionally, on the Zybo Z7 and considered to be the building block of the FPGA are Flip-Flops (FF), Look-up Tables (LUTs), Digital Signal Processing Blocks (DSPs), and Block RAM (BRAM) [13].

Flip-flops are clocked memory devices used in the creation of registers and high-speed sequential memory logic. Flip-flops for a CNN will typically form a pipeline infrastructure with registers between or within layers. This will allow processing between each layer to be performed sequentially while individual node calculations within a layer can be performed in parallel for each clock cycle. In terms of CNNs, any operation able to be

performed via combinational logic could be represented using LUTs. This could be done when performing addition of weight-input products or when performing activation function checks. DSP blocks are typically consumed for multiplications. In a CNN, multiplication of weights with inputs is done for every node in every layer. Pre-programmed weights and input data can be stored in BRAMs. BRAM can also be consumed when any of the above resources are exhausted, performing similar functions but at a slower clock time.

Table I. Zybo Z7 Hardware Components [13]

	Zybo Z7
Flip Flops	35,200
Block RAM	270 KB
Look-up Table	17600

PCAM Integration Attempts and Problems

The Zybo Z7 is capable of capturing and streaming video data directly to the programmable logic on the board with the compatible PCAM device [15]. The video data is fed into the IP cores to decompose the signals into RGB pixel values. We encountered an issue with displaying the output video data when attempting to interface the PCAM with the board. This could have occurred due to connection issues with the PCAM to the board, and/or a faulty PCAM module.

Figure 3. PCAM Output Issues

As this problem was encountered early in the project design, we decided to pivot using the HDMI port for video input instead.

HDMI Demo Attempts and Problems

Instead of using the PCAM for video input, we utilized a laptop with a webcam and HDMI output. We intended for the HDMI output to be directed into the HDMI receive port of the Zybo Z7, where video processing will occur, and an output video feed can be sent from the HDMI transmit port.

Digilent provides an HDMI input/output demo that can be run on the FPGA for easy video passthrough [14]. The demo also features communication with the ARM processor, which will be utilized for image-preprocessing when implementing a CNN on the board. We were successfully able to generate the HDMI demo bitstream and run it on the FPGA. After achieving video passthrough, the next step was to experiment with real-time modification of video data, like what an object classification CNN would do.

We began with generating a grayscale conversion module that accepts RGB values as inputs, and outputs gray-scaled versions of these values. However, when connecting this module to the system, we encountered an issue with bitstream generation. We believe these problems occurred as the HDMI demo utilizes the ARM processor to control timing and communication of video data between the receive and transmit HDMI ports of the board. To fix this issue, we would have to modify C-program files in Vitis and introduce communication between our grayscale module and the ARM processor of the FPGA.

Petalinux Kernel Compilation

AMD provides an embedded Linux platform compatible with a variety of their SoC boards, including the Zybo Z7. This Linux platform will be deployed on the ARM processing

system of the Zybo and will be utilized to run C-programs on the board, that will eventually communicate with the programmable logic of the SoC. The Petalinux OS is provided by AMD as an embedded Linux platform that can be utilized with all Zynq 7000 devices, such as the Zybo Z7 [Appendix A].

After successfully compiling the kernel and placing the Petalinux image onto the SD card, a simple 'Hello World' C-program was written and compiled on the system to ensure functionality.

Vivado/Vitis Install and Setup

Vivado and Vitis are Xilinx's development environments for their FPGAs. Compatible with both Linux and Windows, this software provides a means for us to generate a bitstream from our Verilog code, that can be run on the Zybo Z7 [Appendix B]. Vitis HLS can is also included in the Vivado Suite, capable of generating VHDL code from provided C-files using HLS directives.

Defining a New System

YOLOv3 CNN Implementation with OpenCV

The YOLOv3 (You Only Look Once) is an efficient and popular CNN model for object classification and detection. This model is unique since it applies the entire neural network to the full image and outputs a prediction for regions within the image. Additional context information from the image can be extracted by looking at the image altogether resulting in faster computation time [11]. With a fast model, we initially researched into use of the YOLO network as a new solution for implementing object detection and classification with input video camera data.

We planned to initialize the model on a CPU to achieve a baseline frames per second (FPS) and then compare the FPS to what could be achieved on the FPGA. To start a CPU implementation of the model we downloaded the Darknet detector model onto our host computer system [16]. The Darknet detector model utilizes the YOLOv3 pretrained weights file for processing the image [Appendix C]. With this setup, we were able to test different images in the model and get out prediction accuracies that correlated to the image. As shown below we tested with the image and got the top three results back.

```
data/dog.jpg: Predicted in 23.820582 seconds.
bicycle: 99%
truck: 92%
dog: 100%
```

Figure 4. Yolov3 for Object Classification and Detection [11]

After confirming the model worked with images, we transitioned to modifying the source code to work with a webcam. Interfacing with a webcam required the assistance of OpenCV. We started with downloading the source code, then built the package configurations, and finally installed the application [Appendix C]. Now, with the OpenCV source files installed the darknet model was able to connect to the laptop's camera and begin object detection and classification.

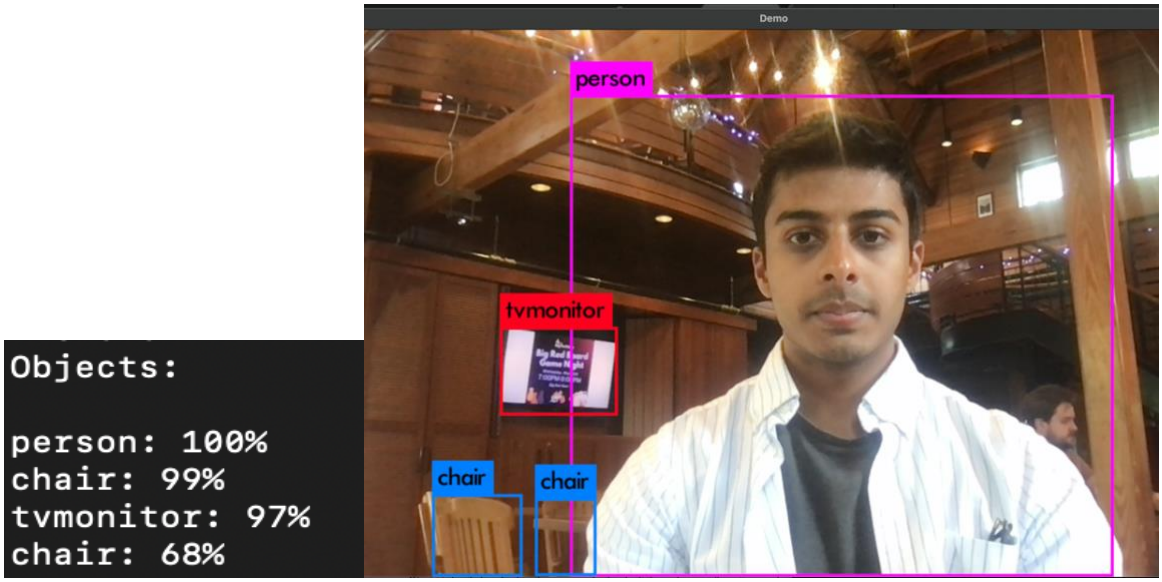

Figure 5. Yolov3 with OpenCv

Getting the webcam, OpenCV, and YOLOv3 to work together highlighted the slow processing speed of a CPU and the potential for acceleration with an FPGA. This process also exposed the memory size of the CNN and video data to perform intensive computation. We attempted to address the CNN memory size issue by implementing the Tiny YOLOv3 model. This is a resource-constrained model provided similar accuracy results with fewer parameters for inferencing. Additionally, we went through the source code to remove extra files not used by the base inferencing function to limit memory issues when transitioning to the FPGA. After our attempts to reduce the size, we had to further research the memory limitation of the FPGA to host our CNN network for a hardware implementation.

Memory Issues

Limited memory resources were one of the key bottlenecks that constrained the improvement of the accelerator's performance and the model we chose to implement. The CNN model's weights and bias were accessed frequently during the inferencing, so we stored all of them on the FPGA. This approach was the most efficient way for the accelerated model to access data for performing computation. However, there were two difficulties in

implementing this strategy on the Zybo Z7. Firstly, the BRAM resources were initially not large enough to store all the data including the model's parameters and feature maps needed for inference. The other difficulty was the original video data captured by the camera also occupied too much footprint space on the FPGA. After realizing the infeasibility of implementing the accelerator on Zybo Z7, we transplanted the neural network model to ZedBoard with more BRAM resources and researched into the use of Tiny Darknet. The pressure on storage resources brought by massive video data will introduce more latency and over utilize memory resources. This sacrifices the performance, which is one of the crucial metrics in a real-time system. Due to memory demands of real-time video processing we chose to change our project goal to static image classification.

Tiny Darknet Convolutional Neural Network

The Tiny Darknet network for image classification highlights the benefits of running a CNN on a FPGA platform. The network is originally written in C++ and uses a series of layers to process the input image, perform computations, and feed the next layer with a feature map. Each layer has been pretrained offline with set parameters used for processing the image and providing the top five output predictions with their associated confidence percentage.

Due to the memory limitation of the FPGA, we used the Tiny Darknet which is a lightweight version of Darknet. The network structure is shown as below:

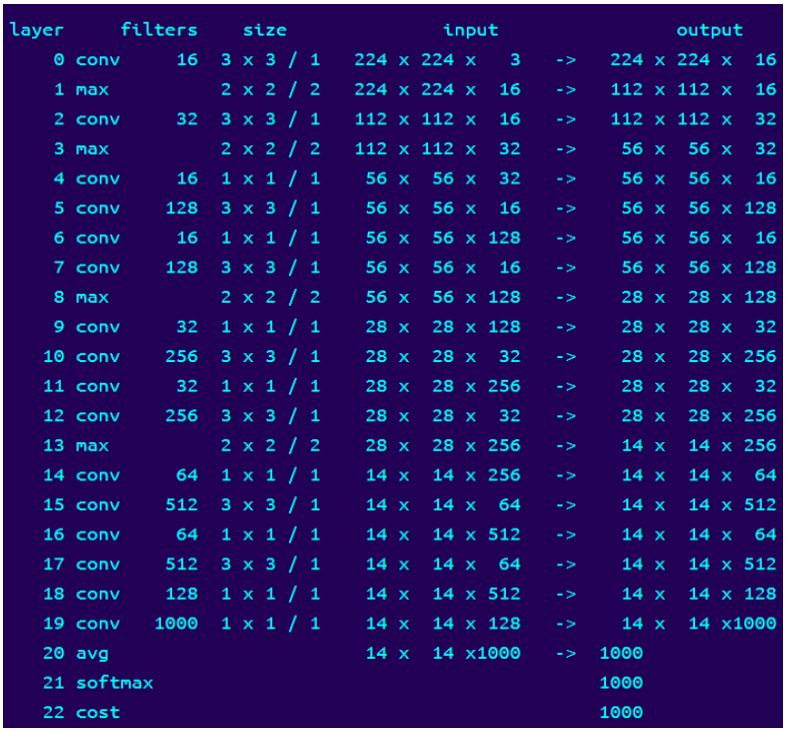

Figure 6. Structure of Tiny-Darknet [11]

Compared to the original Darknet, the size of feature map and the number of layers is reduced, making it possible to be implemented on our board where resource is highly limited. Furthermore, in contrast to other lightweight networks like SqueezeNet, the computational power and inference speed of Tiny Darknet is higher [11].

High-Level Synthesis

High-Level Synthesis (HLS) provides a mechanism for automating the translation from C-level programs to hardware description languages (HDL). This tool is beneficial for developers to create quick hardware implementations of algorithms originally run-on CPUs. We utilized HLS to convert the Tiny Darknet network originally written in C++ to synthesizable RTL that we could generate a bitstream from. HLS requires the C-level function source code, a constraints file, a directives file, and a testbench file as an input for the conversion to HDL [12]. The constrains file outlines the destination hardware board resources and clocking specifications. The directives file contains the optimization directives

that can be used to improve the FPGA's performance compared to the original C-level program. Finally, a testbench file is used to simulate the generated RTL against the C source program output. The tool then outputs the RTL implementation and a report file. The report file provides helpful hardware utilization and latency information that can uncover issues before generating a bitstream for hardware. FGPA hardware over utilization and high latency will cause errors in generating a bitstream and needs to be addressed for a successful deployment.

Figure 7. HLS Conversion

Optimization directives like Pipelining, Unrolling, and Array Partitioning can be included within the C-level program to improve FPGA performance. HLS will then generate synthesizable RTL that takes advantage of the FPGA's parallel architecture based on the implemented optimization directive. We used the Pipelining and Array Partitioning optimization directives in our C-level source code for the Tiny Darknet model. Pipelining allowed us to save on latency cycles in the inner for loops used by the convolutional layer function. Pipelining also allows for multiple iterations of a loop to run at the same time resulting in less clock cycles needed to complete computation. The array partitioning directive was added to the input memory buffers so we could add more read and write ports for storage. HLS allowed us to easily deploy and optimize a CNN model on the FPGA hardware for image classification [12].

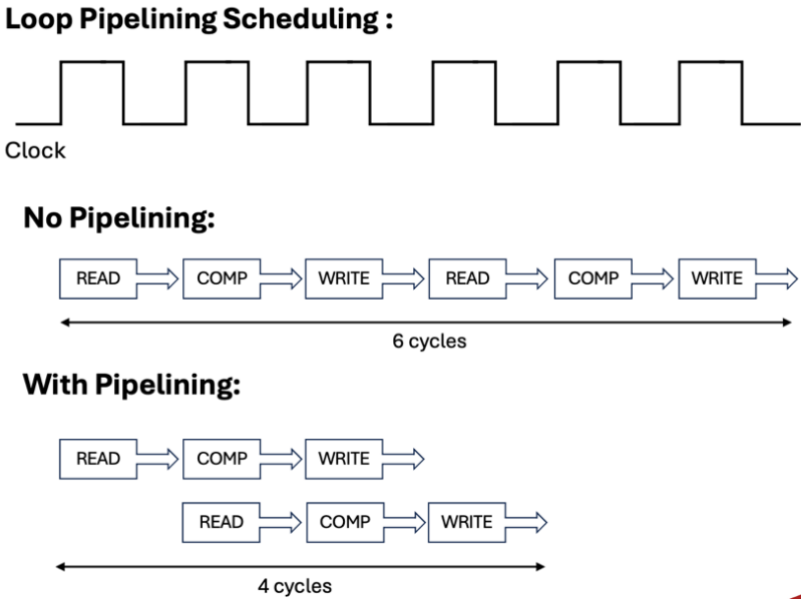

Figure 8. Pipelining Directive [12]

Zybo Z7 to Zedboard

The HLS application that we were utilizing required the Xillybus interface to facilitate communication between the on-chip processing system (PS) and the FPGA programmable logic. The Xillybus interface consists of the Xillinux software that runs on the PS and the IP core that is programmed onto the FPGA fabric. When attempting to initialize the Xillilinux software on the Zybo Z7, it was discovered that this interface was not yet supported on board. The Zedboard Zynq-7000 SoC was compatible with the Xilliinux operating system, motivating our decision to switch to it as our development platform [8] [Appendix D].

Final Solution System Overview

Xillybus

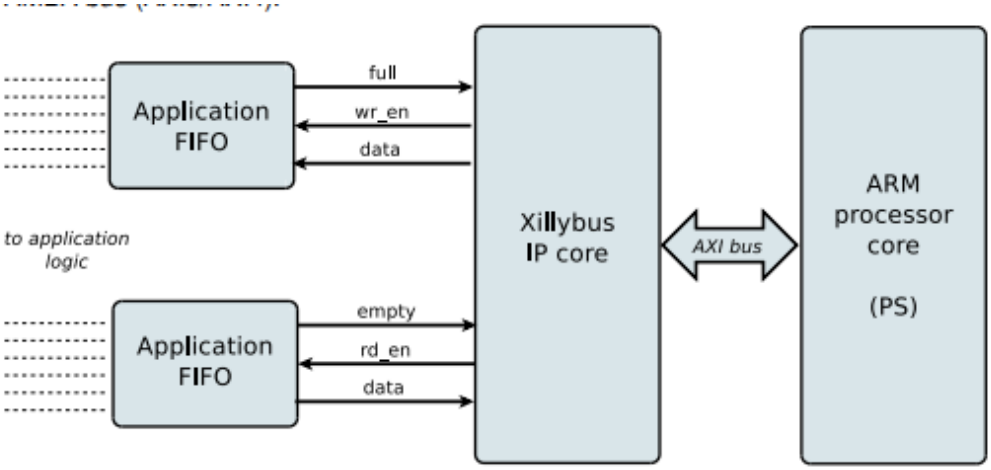

Figure 9. Xillybus Connection [2]

We utilized the Xillybus IP core to facilitate communication between the on-chip processing system and FPGA fabric. The Xillybus offers a simple interface that easily allows for this communication to occur. The Xillybus IP core communicates data between application logic designed by the user and the processing system using application FIFOs shown in the figure above. Reading and writing to the FIFOs is also simple, requiring empty and full control signals along with the data bus.

The ARM processing system interfaces with the Xillybus IP core using Advanced extensible Interface (AXI) communication. For our application, the ARM provides the image data for the FPGA to perform computation over the AXI bus. The data is transmitted between the application FIFOs and the ARM via Direct Memory Access (DMA) requests. Writing to the application FIFOs from the ARM side is viewed as writing to a file/pipe in a typical Linux operating system. After the CNN computation is completed, the results are sent back to the ARM and displayed on the Linux terminal [2].

Translation and Optimization Infrastructure

Translating the C++ network to an equivalent Verilog representation with HLS is not easily done. Certain memory, loop, and data communication optimizations must be made to

ensure that the translated program efficiently utilizes resources on the destination FPGA board. The first problem was the original Tiny Darknet network size was too large to implement on the FPGA. Next, we had to figure out how to transmit data between the ARM and FPGA on just the 32-bit AXI buses. Finally, to achieve a speed up we had to understand the network code and optimize it using the Vivado HLS optimization directives.

To address the network memory size, we reviewed the original network codebase and removed unused functions and old versions. This allows us to properly store the network parameters on the ARM processor. Analyzing the C-level code for parallelization opportunities allowed us to exploit the pipelining directive and reduce latency within convolutional function loops. Parallelization comes at the cost of increased resource utilization. The limited resources of an FPGA need to be taken into consideration when increasing the degree of parallelism. We also added an array partitioning directive to the input memory buffers to allow for more read and write ports for memory. Lastly, converting floating to fixed-point representation allowed for higher efficiency hardware inference because fewer hardware resources are needed to represent data. This resulted in a decrease in BRAM usage but also slightly reduced our prediction accuracy. After converting to 8-bit fixed point representation we could organize the data flow from the ARM to the FPGA by packing four 8-bit data for transmission and optimizing communication for better performance.

Software and Hardware System Overview

Below is a high-level overview of the software and hardware parts of our final system design. Our original network written in C++ was modified with the necessary translation infrastructure so it can be converted into a hardware description language by HLS. After HLS outputs an equal representation in Verilog we can use the Xilinx Vivado compiler to generate

a bitstream file. The FPGA hardware is programmed with a bitstream file generated from the HLS translation. Applications hosted on the FPGA hardware write and read data from the application FIFOs. The host ARM processing system makes Direct Memory Access (DMA) requests via AXI communication to the Xillybus IP core. Upon receipt of requests, the IP core will write or read to the respective application FIFO. The ARM also hosts the network parameters, and the input test images for running inference. The Xillybus IP core creates seamless communication between the ARM processing system and the FPGA programmable logic. This allows for computations to be isolated, performed on the FPGA fabric, and the output results to be displayed using a Linux interface hosted on the ARM.

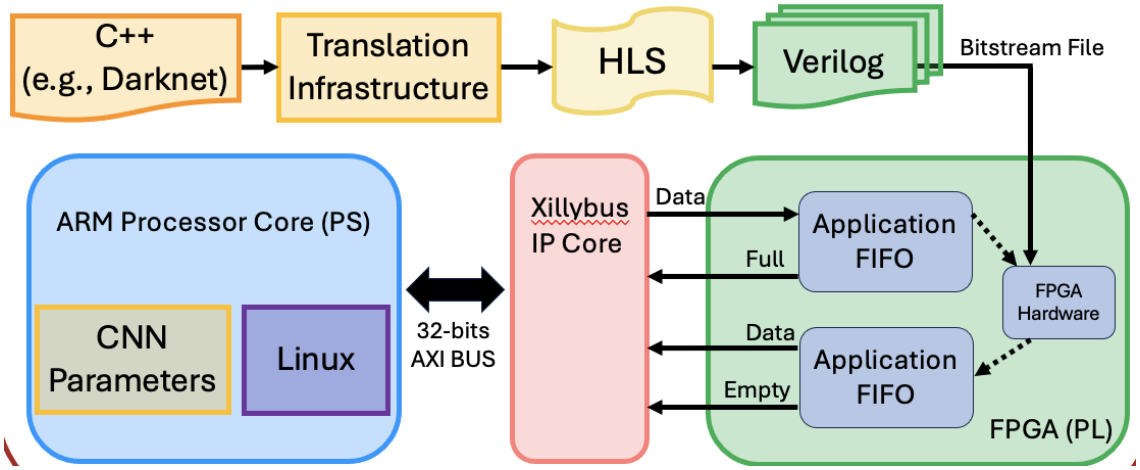

Figure 10. Software and Hardware Overview

Case Study Results: Tiny Darknet

Figure 11. Input Test Image [6]

Accuracy Results

Looking at the inferencing accuracy results we can see that both CNN models correctly predict the input test image of a hummingbird. The CNN network with the optimization infrastructure reduced our accuracy to 50% due to the conversion from floating to fixed point representation.

Table II. Tiny Darknet Inferencing Accuracy Comparison

Inferencing Accuracy		
	Baseline (confidence %)	With Optimization (confidence %)
Hummingbird	**62.18**	**50.20**
Banded gecko	3.23	2.75
Vase	2.66	2.59
Dragonfly	1.97	2.36
Hair slide	1.46	1.87

Prediction Time Results

After implementing the CNN network on the ARM, CPU, and FPGA we achieved a 25x speed from the ARM to the FPGA. The FPGA had the fastest prediction timing due the optimization infrastructure we implemented to efficiently utilize the hardware resources. Additionally, the FPGA has parallel architecture that allows for pipelining unlike the CPU and ARM processor.

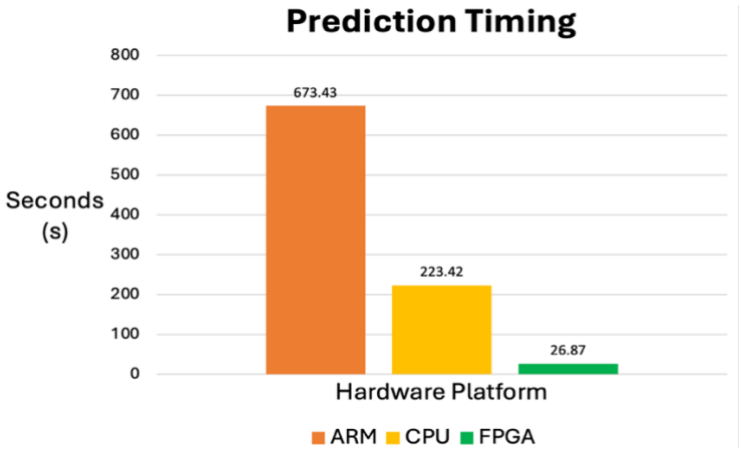

Figure 12. Tiny Darknet Prediction Timing Comparison

Hardware Utilization Results

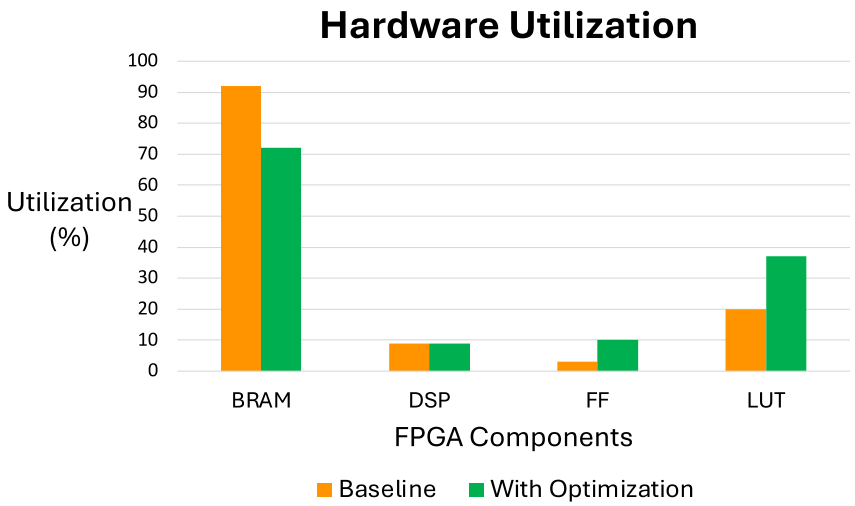

Figure 13. Tiny Darknet Hardware Utilization Comparison

Finally, the hardware utilization from the generated Verilog shows that our optimization reduced the BRAM utilization. Meanwhile the LUT and FF utilization increased. To implement our CNN model, we were bottleneck by memory but our finally implementation fit on the Zedboard with a 73% BRAM utilization.

CNN Acceleration Achievement

We successfully built a process that uses High-Level Synthesis to translate a CNN model written in C++ to Verilog. Our work demonstrates the advantages of implementing computationally intensive algorithms on an FPGA. We achieved a reduction in both prediction time and BRAM memory utilization with the Zedboard FPGA, while still maintaining a high-level of accuracy. The altercations we made to our initial project idea uncovered the different optimizations that must be considered to successfully deploy a model on hardware. Utilizing Vivado HLS optimization directives, a smaller CNN model, and

repackaging information over the AXI bus allowed us to generate improved Verilog code that efficiently used the FPGA hardware.

Future Work

The development time for an application with hardware description languages and an FGPA is significantly more taxing but comes with a beneficial tradeoff of less power consumption. Looking ahead, GPUs are a more flexible hardware resource, but not as efficient as an FPGA and may result in more power consumption. Use of GPUs is still a viable solution that can reduce development time while further enhancing the performance of CNNs.

References

[1] D. Wu et al., "A High-Performance CNN Processor Based on FPGA for MobileNets," 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Barcelona, Spain, 2019, pp. 136-143, doi: 10.1109/FPL.2019.00030. keywords: {Engines;Convolution;Standards;Field programmable gate arrays;Schedules;Acceleration;Computational modeling;convolution neural network;FPGA;hardware accelerator;MobileNet},

[2] “Getting Started with Xillinux for Zynq-7000 v2.0.” Documentation, xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf. Accessed 22 Apr. 2024.

[3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). "Deep Residual Learning for Image Recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

[4] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size." arXiv preprint arXiv:1602.07360.

[5] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). "ImageNet Classification with Deep Convolutional Neural Networks." Advances in Neural Information Processing Systems.

[6] Kulvete, Brian. Ruby-throated Hummingbird Archilochus colubris. 3 May 2018. https://macaulaylibrary.org/asset/97947051. Accessed 22 Apr. 2024.

[7] Lenail, Alex. "NN SVG: Publication-ready NN-architecture schematics." Alex Lenail, https://alexlenail.me/NN-SVG/LeNet.html . Accessed 22 Apr. 2024.

[8] Martha. “Zedboard.” ZedBoard - Digilent Reference, digilent.com/reference/programmable-logic/zedboard/start. Accessed 24 Apr. 2024.

[9] Nash, R. (2015). An Introduction to Convolutional Neural Networks. *ArXiv*. /abs/1511.08458

[10] Pelcat, M., & Berry, F. (2021). Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks? *ArXiv*. /abs/2102.01343

[11] Redmon, Joseph. Tiny Darknet, pjreddie.com/darknet/tiny-darknet/. Accessed 22 Apr. 2024.

[12] “Vivado Design Suite User Guide: High-Level Synthesis.” AMD Technical Information Portal, 4 May 2021, docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[13] “Zybo Z7 - Digilent Reference,” digilent.com. https://digilent.com/reference/programmable-logic/zybo-z7/start

[14] “Zybo Z7 HDMI Input/Output Demo - Digilent Reference,” *digilent.com*. https://digilent.com/reference/programmable-logic/zybo-z7/demos/hdmi (accessed May 14, 2024).

[15] “Zybo Z7-20 Pcam 5C Demo - Digilent Reference,” digilent.com. https://digilent.com/reference/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/star

[16] “4 Steps to Install Darknet with Cuda and Opencv for realtime object detection,” EF Computer, Aug. 19, 2020. https://efcomputer.net.au/blog/4-steps-to-install-darknet-with-cuda-and-opencv-for-realtime-object-detection/ (accessed May 14, 2024).

Appendix A: Petalinux Installation

System Requirement:

- A Linux system is required to complete these steps. A Linux system installed opf Ubuntu 22.04 was used.

Install the necessary libraries and packages.

1. These libraries and packages are listed in the AMD Xilinx documentation and can be found here.
2. Install Petalinux tools.

This tool is what you will use to compile the Petalinux kernel on your Linux system. It also allows you to create new Petalinux projects, or extract and use other Petalinux projects available online.

Run the following commands after downloading installer from the above URL

1. `chmod 755 ./petalinux-v<petalinux-version>-final-installer.run`
2. `./petalinux-v<petalinux-version>-final-installer.run`

Setup the Petalinux working environment.

Then source the Petalinux settings script, making Petalinux specific commands available for use.

1. `source <path-to-installed-PetaLinux>/settings.sh`

Verify that the Petalinux working environment is setup correctly by running the following command. The output will be the path to your Petalinux package.

2. `echo $PETALINUX`

Download and extract Petalinux Board Support Package (BSP) file.

The Petalinux BSP file contains a Petalinux project capable of compiling a kernel compatible with the Zybo Z7 board. It can be found here.

Download the BSP file and run the following commands to source it for a new Petalinux project:

1. `cd <path-to-project-folder>`
2. `petalinux-create -t project -s <path-to-bsp>`

Build the Petalinux project and compile the Petalinux kernel.

The Petalinux kernel will take about five hours to compile. Once completed, the generated images files should be placed on a partitioned SD card to host the Petalinux system.

1. `petalinux-build`
2. `petalinux-package --boot --force --fsbl images/linux/<zynq_type>_fsbl.elf --fpga images/linux/system.bit --u-boot`

Placing Petalinux image on SD card

Partition the SD card:

a. Insert the SD card into your Linux system, then run the following command:

1. `sudo umount /media/<location of mount>`

b. Create the 1st partition on the SD card using the following command:

1. `mkfs.vfat - F 32 -n BOOT /dev/<first partition>`

c. Create the 2nd partition on the SD card using the following command:

1. `Mkfs.ext4 -L rootfs /dev/<second partition>`

d. The second partition must be formatted according to the following prompt responses, in order:

    1. `n - new partition`
    2. `p – primary`
    3. `Partition number: "1" or "2"`
    4. `Partition sector: default (click enter)`
    5. `Partition size: " +<memory size>M/G"`

e. Copy generated image files to the first partition.

    1. `cp images/linux/BOOT.BIN / media/<first partition>`
    2. `cp images/linux/image.ub / media/<first partition>`
    3. `cp images/linux/boot.scr / media/<first partition>`

f. Copy CPIO file containing the "/" directory to the second partition

    1. `cp images/linux/rooftfs.cpio / media/<second partition>`

Following the above steps will generate a bootable Petalinux SD card that can be placed on any Zynq 7000 device, including the Zybo Z7. The next step was to test simple C programs on the Petalinux installation. This will confirm that the embedded Linux software contains the compiler, libraries, and capabilities required for our application.

Appendix B: Xilinx Vivado Installation

The steps outlined in this section can be found here. We worked in a Linux environment for a majority of this project so the instructions shown here will be for a Linux system.

Vivado/Vitis IDE Download and Install

Download Vivado/Vitis install from the Xilinx downloads page, linked here. There versions used for our project include 2023.2 and 2019.1.

Execute the following command after downloading installer:

1. `chmod +x <installer>.bin && sudo ./<installer>.bin`

Enter your Xilinx account credentials when prompted. Select *Vitis* when prompted for product install type. This will install both Vivado (used for interfacing with FPGA fabric) and Vitis (used for interfacing with ARM processor). For compatibility with Zynq 7000 devices (such as the Zybo Z7 and the Zedboard) ensure that 7 Series is selected when configuring installation size. Agree to all terms-of-service prompts, select installation destination, and begin the installation process.

Install Digilent Board Files

Digilent provides board files for FPGA development boards. These board files can be found here in the *new/board_files* directory. Download the board files from the URL and extract the directory. Navigate to the *new/board_files* directory and copy all of the folder located there. Of these copied folders should be a folder named *Zybo Z7-10*. Navigate to the Vivado installation directory, and then to the following directory:

1. `<version>/data/boards/board_files`

Paste the copied directory in the Xilinx installation *board_files* directory. After successful completion of the above, a Vivado and Vitis installation compatible with the Zybo Z7 and all other Zynq 7000 devices will be installed on your system.

Appendix C: Downloading Darknet and Running OpenCV

Installing Darknet

1. `git clone https://github.com/pjreddie/darknet`
2. `cd darknet`
3. `make`

Downloading pre-trained weights

1. `wget https://pjreddie.com/media/files/yolov3.weights`

Testing image file with Yolov3 model

1. `./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg`

Open CV Installation

1. `cd $home`
2. `mkdir OpenCV`
3. `cd OpenCV`
4. `git clone https://github.com/opencv/opencv.git`
5. `git clone https://github.com/opencv/opencv_contrib.git`
6. `mkdir build_opencv`
7. `cd build_opencv`
8. `cmake -DCMAKE_BUILD_TYPE=Release -D OPENCV_GENERATE_PKGCONFIG=ON -DBUILD_EXAMPLES=ON -D CMAKE_INSTALL_PREFIX=/usr/local ../opencv`
9. `make -j7`
10. `sudo make install`

Check packages are properly configured

11. `pkg-config --cflags opencv4`
12. `-I/usr/local/include/opencv4`

Recompile Darknet and edit Makefile

13. Change `opencv=0` to `opencv=1` in Makefile and resave
14. `Make`

Running Darknet with OpenCV

1. `./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights`

C++11 Error

1. Edit Makfile line from `CPP=g++` to `CPP=g++ -std=c++11`

Cannot find opencv.pc Error

1. Check if `opencv4.pc` is at correct location with ls `/usr/local/lib/pkgconfig/opencv4.pc`
2. If not use, `sudo cp /usr/local/lib/pkgconfig/opencv4.pc /usr/local/lib/pkgconfig/opencv.pc`

IplImage Error

1. Add
   ```
   #include "opencv2/core/core_c.h"
   #include "opencv2/videoio/legacy/constants_c.h"
   #include "opencv2/highgui/highgui_c.h"
   ```
   to `/src/image_opencv.cpp`
2. Change `IplImage ipl = m` to `IplImage ipl = cvIplImage(m);`

Appendix D: Xillinux Installation

Steps for installation are summarized here and can be found on this page.

Supported Platforms

- Z-Turn Lite, Zedboard, 7010/20 MicroZed, Zybo (non Z7)
- Vivado 2016+

Required Peripherals

- VGA monitor
- VGA Cable
- USB Keyboard/Mouse

Download Xillinux Distribution

a. Download the Xillinux distribution from the URL located here. Included in the distribution is boot partition kit and Xillinux Image for your partitioned SD card.
b. Unzip the boot partition kit. This consists of various subdirectories containing Verilog, VHDL, TCL scripts and Xillybus IP cores that will be generated into a bitstream and deployed on the FPGA fabric.

Generate Xillybus Bitstream

a. Launch Vivado, click Tools > Run TCL Script and navigate to the downloaded Xillinux distribution directory. Select the *xilldemo-vivado.tcl* script.
b. Select *generate bitstream* and select yes to all prompts that appear.
c. The bitstream file *xillydemo.bit* will be generated at the end of this process. It can be found at the *vivado/xillydemo.runs/impl* directory.

Load the MicroSD Card with Xillinux Image

Note this process is performed on Windows due to complexity of writing image files in Linux.

a. Launch/Install USB image tools.
b. Insert the SD card into your computer. Select *Device Mode* and the SD card that was recently inserted. Select *Restore* and ensure that *Compressed (gzip) image* is the file type that is selected.
c. Navigate to the directory that has the files downloaded from the Xillinux Distribution, and select the image file labeled *xillinux-2.0a.img.gz*.
d. Move the *boot.bin* and *devicetree.dtb* downloaded from the Xillinux Distribution page into the MicroSD card partition labeled *boot*.

e. Copy the xillydemo.bit file generated in the previous steps to the MicroSD card partition labeled *boot*.

   After completing the above, one should have a bootable image of Xillinux available on the SD card. The image can communicate through Linux pipes to the Xillybus IP core and FPGA fabric.