Reconfigurable Real-Time Video Pipelines on SRAM-based FPGAs

Andrew E. Wilson and Michael Wirthlin
NSF Center for Space, High-performance, and Resilient Computing (SHREC)
Brigham Young University
Provo, Utah, USA
{andrew.e.wilson, wirthlin}@byu.edu

***Abstract*—FPGAs are an excellent target for real-time video processing as they provide large amounts of low-level parallelism, low latency, and high bandwidth. However, creating real-time video processing systems on an FPGA is tedious and requires significant effort and low-level digital design skills. This paper presents a technique for creating complex real-time video processing pipelines relatively quickly and easily using partial reconfiguration (PR). A static FPGA system is created that provides a template for a variety of partially reconfigurable video processing cores. A library of video filters has been created that can be inserted into the template regions. At run-time, the user can select the topology of video cores and customize these cores to create complex and unique video pipelines without any understanding of low-level FPGA details. This paper demonstrates this technique with a library of 11 partial reconfigurable regions and 16 video processing cores operating on the Xilinx PYNQ system.**

***Index Terms*—FPGA, Video Processing, Partial Reconfiguration**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are an ideal target for high performance and low latency video processing [1], [2]. They contain a large number of configurable look-up tables (LUT)s, flip-flops (FFs), high-speed I/O, digital signal processing (DSP) units, and block RAMs (BRAMs) to process video data in real-time. Custom video pipelines can be created in configurable hardware to match the computation and throughput of the video processing operations. When using an FPGA, the video processing can be pipelined to maximize throughput with latency defined down to the individual clock cycle. Each stage of these real-time video pipelines contains unique filtering and manipulation functions that together provide complex real-time video functionality. These video processing stages can be implemented in FPGAs as custom hardware circuits to provide low latency video while simultaneously processing video data at high throughput [3], [4]. Creating video pipelines within an FPGA requires the designer to use traditional FPGA design tools such as logic synthesis and FPGA implementation tools. Although the productivity of video processing design can be increased by using FPGA video processing libraries or high-level synthesis (HLS), designing video pipelines on an FPGA is ultimately a low-level, digital hardware design flow. Any changes or customization to an existing FPGA video pipeline must go through the time-consuming FPGA implementation tools.

This paper introduces a method for creating complex video processing pipelines on an FPGA very quickly using dynamic partial reconfiguration (PR). This approach involves the following key components: (1) a static FPGA circuit that contains all static video interfacing logic such as video encoders, decoders, and video memory interfaces, (2) a set of partially reconfigurable regions (PRRs) dedicated for custom video processing modules, and (3) a flexible interconnect to facilitate the communication of video data between video processing cores. At run-time, the user of this system can customize the video pipeline by choosing video filters for the various partial regions and customizing the interconnect between the processing cores. A large number of possible video pipelines can be created in seconds through partial reconfiguration and video pipeline customization.

This approach is demonstrated on the Xilinx PYNQ system that provides interfaces for real-time video, programmable logic, and a host processor system. The PYNQ system has various video sinks and sources including the HDMI interfaces and DDR video direct memory access (VDMA). The programmable logic is organized into a static region to connect to these PYNQ interfaces and 11 different PRRs to hold custom video processing cores. In addition, an AXI interconnect is used to connect these regions to each other for a custom communication topology. A library of cores consisting of 16 different video processing operations is created to fit within the PRRs of the pipeline. A variety of interesting video processing pipelines are demonstrated on this system by configuring the regions with different cores and programming the AXI interconnect appropriately. The video pipeline can easily be changed using a Jupyter notebook running on the PYNQ system.

II. PYNQ Video Sub-System

The run-time reconfigurable video pipeline described in this paper was developed for the Xilinx PYNQ-Z1 system [5]. This system contains a Zynq 7020 device, DRAM, various I/Os and the PYNQ framework. The PYNQ framework includes a large set of pre-programmed hardware functions for I/O interfaces, Ubuntu Linux, Jupyter notebooks, and a Python interpreter. The default PYNQ bitstream or “Base Overlay" is

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1738550.

978-1-7281-1957-1/19/$31.00 ©2019 IEEE

a programmable logic circuit that allows users with no FPGA experience to immediately interact with the board and all its I/O. The PYNQ-Z1 was chosen as the implementation device because of the HDMI I/O, the “Base Overlay" containing a basic video processing pipeline to build from, and a Linux operating system capable of driving the partial reconfiguration.

The base overlay for the PYNQ-Z1 system contains a real-time video pipeline that can sample an incoming video stream and generate an outgoing video stream using the DVI protocol. The video pipeline is implemented in the Zynq's Programmable Logic (PL) and interfaces with the HDMI input port, HDMI output port, and the AXI bus. This low-cost FPGA platform can sample video streams up to 1080p resolution and 60 Hz frame rate. The PYNQ installation provides a number of tutorials for manipulating the video stream using Python code and OpenCV software filters. These examples can be displayed in the Jupyter window or over the HDMI output port.

A high-level overview of the PYNQ video pipeline is shown in Figure 1. This pipeline uses IP blocks developed by Digilent [6] and Xilinx [7]. The “DVI to RGB" module extracts the timing signals from the video display, generates a pixel clock, and converts the incoming DVI signal into a 24-bit RGB color signal. The “Video In to AXI4-Stream" module packages the data and incorporates an AXI4-Stream interface so the video data can be transferred in real-time over the AXI bus. The “AXI VDMA" module provides a DMA engine for transferring the video-in to main memory and transferring video data from main memory to the video-out pipelines. The “AXI4-Stream to Video Out" provides an AXI4-Stream interface for the video data in main memory. The “RGB to DVI" module generates timing signals for the HDMI output and converts the RGB signals into the corresponding DVI output signal. These modules can support monitor resolutions of 1920x1080, 1280x1024, 1280x720, 800x600, and 640x480.

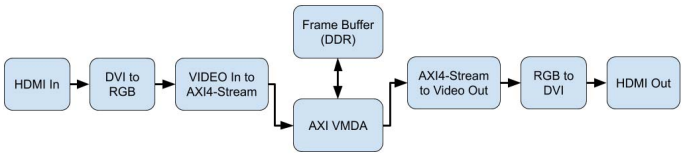

Fig. 1. PYNQ Video Sub-System

The video pipeline can be configured in a number of ways including image frame capture into DRAM, streaming to the HDMI output port, and streaming through the DRAM frame buffer. When the video is streamed through DRAM, custom video processing routines can operate on the video stream in software by the ARM processor. Several software examples within the PYNQ distribution demonstrate the ability to capture input video data, perform software-based image processing, and then stream the result to the HDMI output port. These examples, written in Python and linked to an OpenCV library, include: changing RGB values, edge detection, and face detection. Running in software, these examples are not able to run at the full 60 Hz frame rate, but run at around 2-3 frames per second due to the limitations of processing the video in the processor. This video pipeline provided on the default PYNQ overlay provides a convenient and easy way to experiment with real-time video streams and learn about video systems.

III. Programmable Video Pipeline

Although the default video pipeline on the PYNQ system provides the ability to capture and produce real-time video data, it is not organized to easily support custom hardware circuits using the available PL. If custom video pipelines are needed, the designer must modify the design of the PL and resynthesize that design using vendor tools. Those interested in making custom video pipelines must have the expertise and tools necessary for modifying the logic circuits and correctly implementing this logic on the PYNQ system.

This paper introduces a method to simplify the deployment of custom video pipelines without the need to redesign the logic of the PYNQ video pipeline. This approach relies on partial reconfiguration to reconfigure various regions of the FPGA with precompiled partial bitstreams that implement discrete video processing functions. By interconnecting these video functions at run-time, complex and unique video pipelines can be configured with little understanding of the digital logic used to implement each function. Furthermore, the video pipeline can be changed at run-time to implement a variety of different video pipelines on the same PYNQ system. This work is an extension of a class project that demonstrated the use of a single PRR for simple video functions [8]. The single PRR operated on the raw VGA video signals and implemented filters defined with hardware description language (HDL) code.

Other works have shown effective, FPGA-implemented video processing cores capable of handling live video at 600×800 pixel resolution with a 60 Hz frame rate. These cores included Harris Corner [9], Sobel, Robert, Prewitt, and Laplacian filters [10]. The Canny edge detector has been implemented in the PYNQ hardware that is capable of the full 1080p bandwidth at 60 frames per second [11]. Additional works have demonstrated the benefits of using dynamic PR for video processing cores to save resources and add configurability to the video processing pipeline [12]–[15]. One of these dynamic reconfigurable pipelines was able to process 720p video at 60 frames per second [16].

The dynamically reconfigurable video processing pipeline presented in this paper is composed of two distinct components: a static logic circuit and a library of partially reconfigurable video processing functions. The static logic circuit contains the fixed logic for the video I/O interfaces, dedicated “stubs" or PRRs for the reconfigurable video circuits, and an interconnect network to customize the communication between circuit functions. The library of video processing functions each implement a distinct function that when combined with other functions can produce interesting and complex video processing pipelines. Complex video processing pipelines can be created without the long process of re-implementing the

whole bitstream. The following two sections will describe the static video processing circuit and the library of video functions.

IV. Static Video Processing Framework

The first component of the programmable video processing pipeline is a static logic circuit that provides the fixed functionality of the video pipeline and the fixed framework for partially reconfigurable video processing units. This logic never changes and is configured once at boot time in the PYNQ system. This static circuitry provides the following three essential functions: (1) static interfacing logic, (2) reconfigurable regions, and (3) interconnect for the reconfigurable regions. An overview of this static framework is shown in Figure 2.

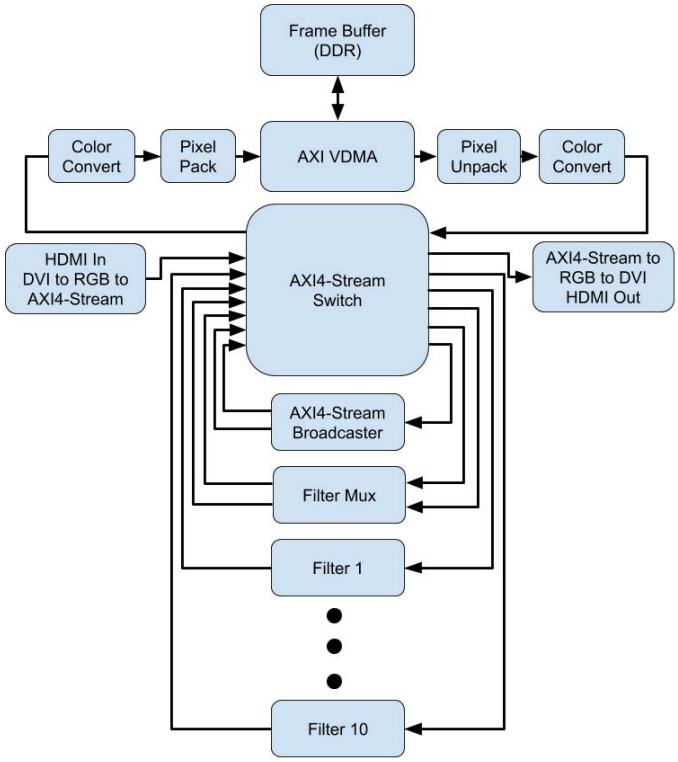

Fig. 2. Static Video Processing Circuit

A. Static Logic

The static interfacing logic defines all the circuitry of the FPGA digital design that is consistent for any possible configuration. There are a number of important functions that must be provided within the static logic to support the operation of the video processing pipeline. This static logic contains the main AXI-Lite bus connected to the ARM processor and allows the processor to interact with the PL. All of the I/O interfaces such as LEDs, switches, and HDMI interfaces also remain in the static logic.

The video processing pipeline is split between the static and dynamic logic. Although the PYNQ base overlay is not used in this project, the logic used to create the video sub-system within the PYNQ base overlay is used within the static region of this video processing system. As shown in Figure 2, the components of the PYNQ video sub-system have been included in the new pipeline. This includes the conversion of the HDMI input to RGB data stream over AXI4-Stream and the conversion for the HDMI output. The original video sub-system includes video processing cores to perform color conversion and data packing in the PL. These cores are included in the static logic around the video DMA interfaces to the DDR memory. The video pipeline clock did not change from the 142 MHz in order to process real-time video of up to 1080p resolution and 60 Hz frame rate.

B. Video Processing Template

In addition to the static logic, basic video processing templates were instantiated as black boxes to later be defined by the dynamic logic. The video processing cores implemented in these templates operate on the data within the AXI4-Stream interfaces similar to the “Color Convert" processing units found in the original PYNQ base overlay. The AXI4-Stream interface can deliver 24-bits of RGB video data, a flag for the start of the frame, and a flag for the end of a line within the frame every clock cycle. The basic template for the dynamic video processing logic includes two AXI4-Stream interfaces as input and another as output. Additional inputs were added to allow the processor to configure video processing core-specific parameters during run-time. Ten of these templates were added to the design to allow for ten separate video processing cores.

C. Video Stream Mixing

Besides the basic video processing template that uses one input and one output stream, two additional modules were chosen. The first module is a Xilinx AXI4-Stream Broadcaster that takes one input stream and broadcasts it to two output streams. This allows for more complex video pipelines or for the final processed video to be broadcast to both the DDR and the HDMI out-port. This module operates on the video within the static logic of the pipeline.

The second module is a special video processing template, the “Mux". This module takes two inputs, mixes them according to the dynamic logic provided, and outputs to one or both of the available AXI4-Stream outputs. This provides the useful function to combine two different video processing streams into one stream. This unique template could be used for applications such as a “green screen" filter, split screen, or the absolute difference between the incoming streams. Only the “green screen " filter was implemented in this pipeline.

To guarantee that two input video streams are synced for this module, an additional custom HDL solution was added to this special template. The custom HDL stalls a non-real-time video stream driven by the VDMA output until the frame start flag of the AXI4-Stream is aligned with the real-time video stream.

D. Partially Reconfigurable Regions

The PRRs define the configurable FPGA resources dedicated to the dynamic logic that will be configured during run-time. The physical locations must be specified for each PRR to separate them from the static logic. Through several iterations of implementing different sizes for the PRRs, three different sizes where chosen for the 10 basic templates to allow for the greatest number of PRRs and support video processing cores of various resource utilizations. Six small PRRs were created for use by the video processing cores that required few resources such as LUTs, FFs, DSPs, and BRAMs. Some of the video filters used within this design required additional logic resources to be properly implemented within the bounds of a PRR. Three medium PRRs were added to the design for these video filters. One large PRR was created for video cores that required a significant amount of BRAMs. Any core that can be implemented in a small PRR can also implemented the medium or large PRR, and any medium PRR filter can be implemented in the large PRR. The “Mux" has a unique PRR because it is a template with a different number of I/O interfaces and can not be exchanged with the other video processing cores in this design. The “green screen" filters implemented within this template does not require many resources in this implementation and is given the smallest PRR.

The placement of these PRRs was based upon the location of special resources such as BRAMs and DSPs while meeting the timing constraints of the HDMI input and output. The Vivado floorplan in Figure 3 shows the placed and routed static logic and the resources dedicated to the PRRs. Although the static region contains logic that does not change, most of the programmable logic on the system is reserved for run-time configurable logic. Table I shows the utilization of the static region and the resources each PRR has available to implement the video processing cores.

TABLE I
COMPARISON OF PR REGIONS

PR Region Utilization				
Region	LUT	FF	BRAM	DSP
Static	14434	24582	15	12
Large	5200	10400	20	20
Medium	3200	6400	10	20
Small	2400	4800	10	20
Mux	1200	2400	0	0

Each PRR was instantiated as a black box in the digital design prior to synthesis. Once synthesis was completed, Xilinx pblocks were assigned to each black box to define the resources and physical placement of the PRRs. Video processing cores were temporarily added to allow the tools to account for average timing when the design was implemented. After the design was placed and routed, the video cores were removed and the design was locked down to prevent future partial designs from affecting the static region, as depicted in Figure 3. A TCL script was generated to automate this process for future changes to be added to the static design.

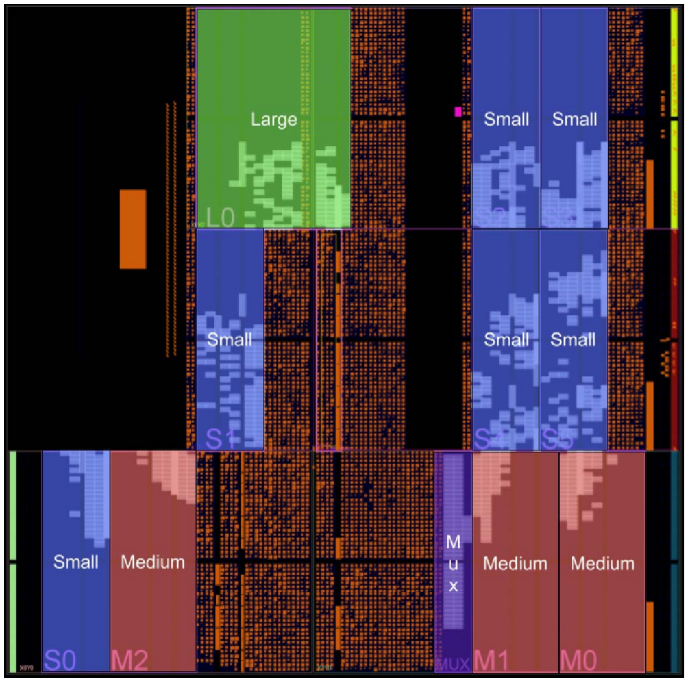

Fig. 3. Floorplan

E. Pipeline Interconnect

This video processing pipeline allows multiple video streams to be processed, mixed, and/or copied with the use of multiple video sources, the “Mux" processing template, and the AXI4-Stream Broadcaster. To provide the high amount of configurability to support a variety of different processing paths, a Xilinx AXI4-Stream Switch is implemented in the static logic. This provides an interconnect between the different components of the static and dynamic logic. The switch allows for the controlled flow of 16 input and output streams configured by registers that can be accessed by the processor over an AXI-Lite bus. This dynamic switch adds an additional two cycles of delay between every connection.

Figure 2 shows the layout of the AXI4-Stream Switch that directs the video stream between the static logic and the dynamic video processing cores. The HDMI input and output stream are connected directly to the switch. This requires that the pixel clock be passed from the input to the output in the case of the “AXI VDMA" not being included in the pipeline to act as a frame buffer. The “AXI VDMA" was also included as an optional port on the switch with the original PYNQ Xilinx modules that allow for color conversion and data packing to optimize the use of the DDR and processor. These video sources and sinks can be pipelined with the ten basic processing templates, the “Mux", and the AXI4-Stream Broadcaster available on the switch interconnection.

V. PARTIALLY RECONFIGURABLE VIDEO FILTERS

The second primary component of this programmable video pipeline system is the library of partially reconfigurable video processing cores. These cores are designed to operate within

the PRRs defined in the static region. Most of the video filters operate on a single real-time video stream and produce a modified video stream (filters designed for the 'Mux' reconfigurable region operate on two video streams). The incoming video stream includes 24-bits of RGB data, a flag for the start of the frame, and a flag for the end of the line within the frame.

A variety of video processing cores of different functionality and resource utilizations were developed to demonstrate this reconfigurable video processing pipeline concept. The following cores were included in this work:

- **Pass:** Passes data through with no changes.
- **Threshold:** Performs a threshold function that converts each pixel into a binary value.
- **Color Limiter:** Limits the max color value for each of the three RGB channels.
- **Draw Lines:** Draws four lines over the image, two horizontal and two vertical, with variable locations and widths.
- **Invert:** Inverts the color values of each pixel.
- **Grayscale:** Calculates the grayscale value for each pixel and outputs to all three channels.
- **Mirror:** Mirrors the image from left to right.
- **Emboss:** Applies an embossing algorithm on the full image.
- **Erode:** Performs the erosion morphological operation.
- **Dilate:** Performs the dilation morphological operation.
- **Sobel:** Performs Sobel edge detection.
- **Kernel 3x3:** Applies a parameterizable 3x3 kernel filter to support custom operations such as sharpen, outline, and blur.
- **ASCII Overlay:** Overlays ASCII characters on screen with variable size and location.
- **Hirigana Overlay:** Overlays Japanese Hirigana characters on screen with variable size and location.
- **Image Overlay:** Applies a static image stored in BRAM on the image at a parameterizable location.
- **Green Screen:** Segments the image into a foreground and background based on a parameterizable range of RGB values.

The library of filters can easily be extended by designing other filters that fit within any of the PRRs and meet the interfacing requirements.

The video processing cores are designed using C++ and synthesized using HLS. Each core was fully pipelined with an iteration interval of one, allowing the function to begin processing one pixel every clock cycle. Some filters required video line buffers to allow for more complex functions that required additional video frame information. Other filters required BRAM memory to store dynamic images and static fonts that were overlaid on the video image. Each video core used the AXI4-Stream for video data to simplify the design of the interfacing logic as the Vivado HLS tools automatically handled the acknowledgement protocol for the AXI4-Stream interface.

Each filter was mapped to as many of the PRRs as possible to maximize the flexibility of building complex pipelines at run-time. Those filters that fit within the *Small* regions are mapped to each *Small*, *Medium*, and *Large* region within the static design. Filters fitting in the *Medium* regions are mapped to the *Medium* and *Large* regions. The build process for generating this set of bitstreams involved HLS synthesis, out-of-context HDL synthesis, and then placement, routing, and bitstream generation for each potential PRR. A custom TCL script was written to allow for the automated implementation and generation of the partial bitstreams.

Table II summarizes implementation time of creating bitstreams for all 16 filters using a computer with a i7-4770 CPU at 3.40GHz and 16GB of RAM. The first row also includes the time to implement one static pipeline that includes a large set of these filters. Generating individual filter circuits is much faster than generating a full static design, suggesting that faster to generate partial circuits for the complex pipeline than to create a custom static pipeline for each pipeline variation.

TABLE II
COMPARISON OF IMPLEMENTATION TIMES

Designs	Time (min)
Static	62.94
Pass	11.87
Threshold	12.10
Color Limiter	11.75
Draw Lines	13.45
Invert	11.53
Grayscale	11.90
Mirror	12.13
Emboss	13.13
Erode	12.91
Dilate	12.81
Sobel	18.17
Kernel 3x3	17.54
ASCII Overlay	16.97
Hirigana Overlay	17.35
Image Overlay	15.03
Green Screen	13.49

The resource utilization of each of the filters is summarized in Table III. This table also includes the latency in clock cycles of each filter as well as its potential PRR placement (small, medium, and/or large). The video filters of Mirror, Erode, Dilate, Sobel, and Kernel 3x3 use line buffers to perform their operations and require additional latency to fill the buffers.

VI. VIDEO PIPELINE OPERATION

Once the bitstreams have been generated for the PRRs and uploaded to the PYNQ file system, custom video pipelines can be created. Python classes and Juypter interfaces were created to automate this process for the user. The process for creating a custom pipeline is as follows:

1) The partial bistreams are first configured into the appropriate FPGA region using the `xdevcfg` drivers to access the the processor configuration access port (PCAP).
2) Each filter can be customized with run-time specific parameters (such as filter coefficients, text strings, etc.) using a custom control bus. The user can change these settings at run-time using the Jupyter interface to customize the pipeline as needed.

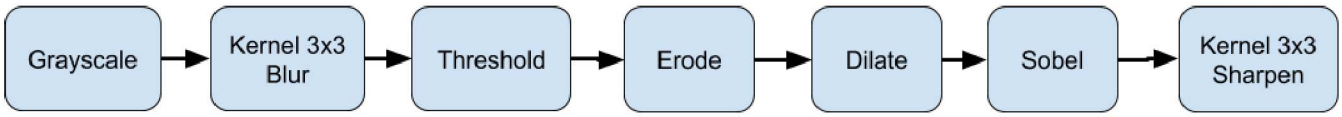

Fig. 4. Edge Detection Video Processing Example

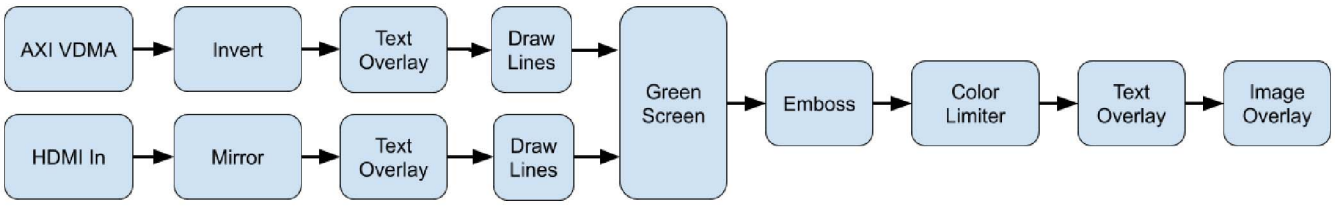

Fig. 5. Green Screen Video Overlay Example

TABLE III
COMPARISON OF HLS VIDEO FILTERS

Filters	LUT	FF	BRAM	DSP	Latency	PRR
Pass	70	130	0	0	2	S,M,L
Threshold	117	274	0	1	6	S,M,L
Color Limiter	127	263	0	0	3	S,M,L
Draw Lines	547	579	0	0	5	S,M,L
Invert	70	130	0	0	2	S,M,L
Grayscale	91	172	0	1	6	S,M,L
Mirror	121	172	3	0	4*	S,M,L
Emboss	175	232	3	0	6	S,M,L
Erode	217	533	1.5	1	8*	S,M,L
Dilate	193	445	1.5	1	8*	S,M,L
Sobel	1254	3978	1.5	6	51*	M,L
Kernel 3x3	1297	3222	1.5	10	51*	M,L
ASCII Overlay	545	1346	1	0	11	S,M,L
Hirigana Overlay	671	1364	0.5	0	11	S,M,L
Image Overlay	479	1166	16.5	0	9	L
Green Screen	295	533	0	0	4	MUX

3) The topology of the communication between video filters is set by configuring the AXI4-Stream Switch.

Once these steps have been completed, the custom video pipeline is ready to process real-time video streams.

A. Python Pipeline API

For the PYNQ to support the configurable video processing pipeline, additional Python functionality is needed. Using the "MMIO" class provided by the PYNQ Python API, a subclass was created to configure the AXI4-Stream Switch and send commands to each filter. The "MMIO" class allows Python to memory map Linux protected memory and have read and write access to these addresses. The AXI4-Stream switch and CMD Bus were connected to the main AXI-Lite interface of the processor. The Python class PRControl used a map to simplify the process of configuring the AXI4-Stream Switch.

Listing 1 shows a Python example of configuring the video processing pipeline with a single color limiter core. One of the small PRRs is reconfigured with the correct partial bitstream on line 2. The PRControl class also has a function to send commands to each filter. The function, filter_cmd, takes the name of the filter, the address of the variable, and the data. Line 4 shows an example usage of this function to set the maximum red color value in the color limiter core. The command on line 6 connects the hdmi input to PRR S0 by writing to the control registers of the AXI4-Stream Switch. Lines 8 and 10 connect that S0 PRR to the VDMA to allow the processor access the video data over the AXI-Lite bus, and forwards the VDMA to the HDMI output, displaying the final frame on an external display. The final output will show the processed image limiting the red portion of the pixels to a max value of 0xA0.

Listing 1. Pipeline Contols

```
1  # Reconfigures S0 PRR with the color limiter core
2  PartialBitstream("limit_s0.bit").download()
3  # sets register 0 of S0 PRR to 0xA0
4  prcontrol_i.filter_cmd("S0",0,0xA0)
5  # Connects the live HDMI to S0 PRR
6  prcontrol_i.connect("HDMI_IN","S0")
7  # Connects S0 PRR to the DDR VDMA frame buffer
8  prcontrol_i.connect("S0","VDMA")
9  # Connects the DDR VDMA frame buffer to the HDMI output
10 prcontrol_i.connect("VDMA","HDMI_OUT")
```

B. Example Video Pipelines

A number of unique video pipelines have been created on the PYNQ system using this static video region and the library of video filters. Two examples will be described here: an edge detection pipeline and a "green screen" mixing pipeline. The topology of the edge detection pipeline is shown in Figure 4. This pipeline includes the following filters organized into a linear sequence of video processing steps: grayscale, 3x3 kernel (using a blur configuration), threshold, erode, dilate, Sobel filter, and a 3x3 kernel (using a sharpen configuration). Several of these filters require parameters that can be set by the user to create a very effective real-time video edge detector.

A second example is a "green screen" video overlay as shown in Figure 5 . This pipeline uses all 11 filter slots and mixes two real-time video streams. The first stream is generated from an image in DDR memory using the AXI VDMA channel and undergoes the following steps: invert, text

overlay (with a user customizable message), and line insertion. The second video stream is driven by the HDMI input port and is mirrored and augmented with both custom text and lines. The “green screen" mux filter identifies regions within the incoming video stream that fit within a range of color values (i.e., "green screen" values) and then replaces these pixels with those of the background image. The resulting video stream then undergoes the following final steps: emboss, color limiting, text overlay, and small image overlay.

Each of these two sample pipelines were easily created in a matter of seconds by a user interacting with the PYNQ Jupyter pages. Many other interesting video pipelines can be created by organizing these library functions in other unique topologies.

C. Dynamic Pipeline Specifications

The video processing pipeline consists of one full bitstream, 15 large partial bitstreams, 42 medium partial bitstreams, and 72 small partial bitstreams. All of these sum to to 131 bitstreams consisting of 64137 KB that can create billions of possible hardware configurations using the small, medium, and large PRRs[1]. The average times to reconfigure the large, medium, and small PRRs are 50.5, 32.8, and 30.9 milliseconds respectively, using the `xdevcfg` Linux driver. The PYNQ system can reconfigure the whole pipeline in under a second with any configuration. All the possible configurations support the full 1080P video at 60Hz bandwidth. The max cycle latency of the pipeline, ignoring any necessary row buffering, is 600 clock cycles of a 142 MHz clock or 4.23 μs. The power utilization ranges from 2.063 W for the empty static design to 2.515 W for a fully loaded design.

This configurable video processing pipeline tested the limitations of the PYNQ with 11 different PRRs and 16 various video processing cores. The PYNQ Jupyter demonstration files can be found in this public repository[2]. The hardware and TCL build scripts are available at this repository[3].

VII. Conclusion

This paper describes a unique approach for generating custom video pipelines in real-time by reconfiguring the FPGA with pre-built video processing filters. This approach involves creating a static circuit with fixed I/O interfaces, reconfigurable regions, and a programmable interconnect. In addition, a library of video processing cores was created in C++ using HLS and then mapped to each of the partial reconfigurable regions. This approach significantly reduced the implementation time needed to create a custom video pipeline. Custom pipelines can be organized in a matter of seconds rather than the hours required to fully implement a custom pipeline using traditional FPGA tools.

A number of future enhancements to this work are being pursued. First, this work is being upgraded to the recently released ZCU104 PYNQ system that supports higher performance quad-core processors and more FPGA configurable resources. Further, this system uses the more advanced and efficient UltraScale+ FPGA architecture using FinFET technology. Second, additional and more complex video filtering circuits are being investigated including the Xilinx xfOpenCV HLS library based upon the OpenCV software library.

[1] $12^6 \times 14^3 \times 15 = 1.23 \times 10^{11}$ possible combinations

[2] https://github.com/byuccl/BYU_PYNQ_PR_Video_Pipeline

[3] https://github.com/byuccl/BYU_PYNQ_PR_Video_Pipeline_Hardware

References

[1] M. Genovese and E. Napoli, “ASIC and FPGA implementation of the gaussian mixture model algorithm for real-time segmentation of high definition video,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 3, pp. 537–547, March 2014.

[2] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, “FPGA-Based real-time pedestrian detection on high-resolution images,” in *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, June 2013, pp. 629–635.

[3] A. Tumeo, S. Borgio, D. Bosisio, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, “A multiprocessor self-reconfigurable JPEG2000 encoder,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–8.

[4] G. A. Vera, D. Llamocca, M. s. Pattichis, and J. Lyke, “A dynamically reconfigurable computing model for video processing applications,” in *2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, Nov 2009, pp. 327–331.

[5] “Python productivity for zynq.” [Online]. Available: http://www.pynq.io/

[6] Digilent. (2019) Digilent vivado library. [Online]. Available: https://github.com/Digilent/vivado-library

[7] Xilinx, “Xilinx ip catalog,” 2019.

[8] B. Hutchings and M. Wirthlin, “Rapid implementation of a partially reconfigurable video system with PYNQ,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[9] E. Onat, “FPGA implementation of target detection algorithm at real time video signal processing using harris corner detector filter,” in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, May 2018, pp. 1–4.

[10] ——, “FPGA implementation of real time video signal processing using sobel, robert, prewitt and laplacian filters,” in *2017 25th Signal Processing and Communications Applications Conference (SIU)*, May 2017, pp. 1–4.

[11] B. C. Maheshwari, J. Burns, M. Blott, and G. Gambardella, “Implementation of a scalable real time canny edge detector on programmable SOC,” in *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, Nov 2017, pp. 1–5.

[12] R. Khraisha and J. Lee, “A scalable h.264/avc deblocking filter architecture using dynamic partial reconfiguration,” in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, March 2010, pp. 1566–1569.

[13] L. S. U. Rani, G. Jagajothi, and P. T. Selvan, “Digital filter for real-time impulse noise suppression in video processing using dynamic partial reconfiguration technique,” in *2015 International Conference on Control Communication Computing India (ICCC)*, Nov 2015, pp. 433–436.

[14] S. U. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan, “Real time video processing on FPGA using on the fly partial reconfiguration,” in *2009 International Conference on Signal Processing Systems*, May 2009, pp. 244–247.

[15] S. Bhandari, S. Subbaraman, S. Pujari, F. Cancare, F. Bruschi, M. D. Santambrogio, and P. R. Grassi, “High speed dynamic partial reconfiguration for real time multimedia signal processing,” in *2012 15th Euromicro Conference on Digital System Design*, Sep. 2012, pp. 319–326.

[16] M. Nguyen and J. C. Hoe, “Time-shared execution of realtime computer vision pipelines by dynamic partial reconfiguration,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 230–2304.