Partial Reconfiguration of FPGA Image Processing Pipelines with PYNQ

Tim Häring

August 20, 2021

This project implements a framework to utilize partial reconfiguration of FPGAs on a high abstraction level with PYNQ, which allows software developers to easily benefit from hardware acceleration. With the help of image processing IP-cores generated with HLS, a prototype algorithm is accelerated by a factor of about 15. The approach using dynamic partial reconfiguration also utilizes only about half the resources a traditional hardware design would require while offering a greater amount of functionality, highlighting the benefits of this methodology.

1 Introduction

Image processing and computer vision have become pervasive in modern society with applications ranging from robotic vision, object detection and identification to autonomous driving and many more. With an increasing number of applications and use cases, the complexity and required computation power of systems also rose [4]. Many desktop and server applications are able to use the processing power provided by Graphics Processing Units (GPUs) and countless frameworks and Application Programming Interfaces (APIs) are available to ease development, whereas Field Programmable Gate Arrays (FPGAs) stood out for applications with tight power constrains and highly parallel algorithms, though limited by the complexity of the configuration process [5]. This work aims to combine several approaches that ease FPGA programming for image processing pipelines with the help of different abstraction layers and computer vision libraries while simultaneously overcoming the hard size restrictions that limit the amount of computation that can be carried out on a FPGA.

1.1 Image Processing

For many years researchers have tried to mimic the human visual system in order to create robots and digital system with intelligent behaviour. With functions ranging from facial and gesture recognition to object detection and motion tracking, the Open Source

Computer Vision Library (OpenCV) [2] is one of the most comprehensive image processing libraries that currently exists. Besides providing several hundreds of computer vision algorithms, it also allows researchers to easily experiment and combine different approaches due to its modular structure and simple API. Due to the widespread application of image processing in a multitude of systems and the similarities of algorithms that operate on video and image data, this field was chosen to create a proof of concept framework. With minor tweaks the resulting framework could also be utilized in vastly different contexts.

1.2 FPGA Programming

As mentioned previously, creating efficient and correct FPGA programs is no simple manner and requires thorough system and device knowledge. Xilinx, one of the largest manufacturers of FPGA devices, and many other commercial vendors, as well as academic researchers, attempted to provide simpler approaches to hardware programming to open up the possibilities and acceleration capabilities of these devices to a wider audience [9]. Figure 1 illustrates different abstraction layers and how they relate to the traditional programming approach.

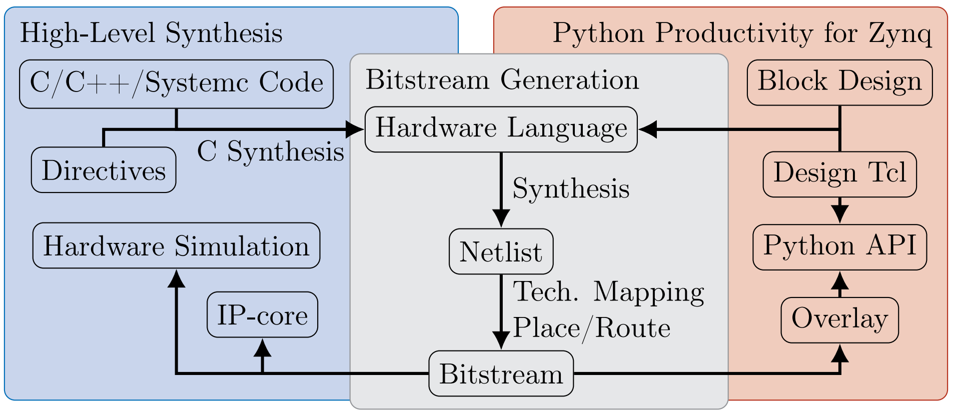


Figure 1: Different FPGA programming abstraction layers.

While the conventional approach of bitstream generation requires users to define the hardware they would like to create in a Hardware Description Language (HDL), High-Level Synthesis (HLS) and Python Productivity for Zynq (PYNQ) provide abstraction layers in high-level languages that are familiar to software developers.

1.2.1 High-Level Synthesis

Using HLS for FPGA development allows for easier and faster testing of functional correctness, portability of code and shorter development cycles. Vivado HLS from Xilinx uses pragmas to annotate C/C++/SystemC code with compiler hints in order to produce efficient hardware. This way developers do not need to learn a HDL or Domain-specific Language (DSL) but creating complex systems still requires profound knowledge of the underlying hardware. Because of the huge popularity and wide spread usage of the OpenCV library, Xilinx provides an equivalent HLS implementation of most of the functionality annotated with appropriate pragmas to produce effective circuits, which was used throughout the curse of this project.

1.2.2 Python Productivity for Zynq

PYNQ provides software developers the opportunity to easily access and utilize custom circuit functionality and to collaborate closer with hardware engineers. It allows for reusage of carefully crafted hardware definitions through an abstract and simple Python API. It can only be used on the Xilinx systems that combine a Processing System (PS) and Programmable Logic (PL) into one System-on-Chip (SoC). The PS runs a custom Linux operating system that provides a Jupyter-Notebook server for quick and easy access to the system. Python is then used to control the PS and perform calculations in software, as well as to specify the hardware building blocks and delegate computation to the custom circuits implemented by the selected overlay. With this dynamic approach, computation offloading can be easily tested and evaluated without any hardware development knowledge.

1.2.3 Partial Reconfiguration

Where Central Processing Units (CPUs) and GPUs can simply be tasked to carry out specific calculations by executing machine code, FPGAs can only ever execute the algorithms that were synthesized to the bitstream that was used to program the device. Since the amount of reconfigurable elements on an FPGA is limited, this poses a hard limit to the number of accelerators that can be executed on the FPGA. When another functionality is required, the entire FPGA needs to be reconfigured with a new bitstream that needs to be synthesized first. As bitstream creation (one hour or more) and FPGA reconfiguration (several seconds or more) is very costly time wise, even live reconfiguration is not feasible in an image processing context. To circumvent this limitation, Xilinx provides Dynamic Function eXchange (DFX), which allows for the reconfiguration of modules within an active design. Figure 2 illustrates this behaviour.

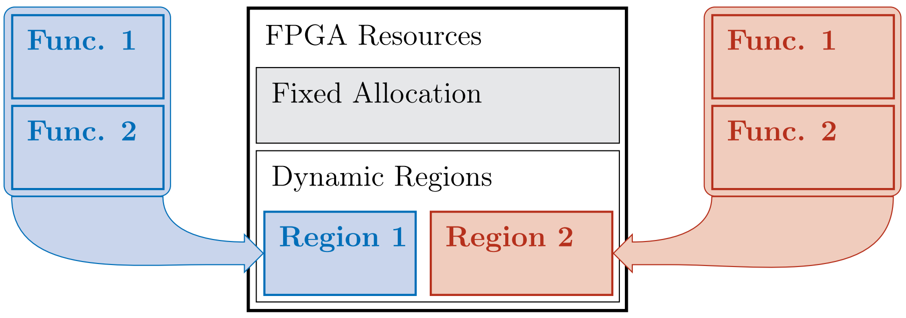


Figure 2: FPGA resource usage with partial reconfiguration.

While not being able to completely negate the downside of limited reprogrammable fabric, partial reconfiguration offers several advantages. As stated in [6], it allows for increased system performance since swapping functionality does not require downtime, hardware sharing for reduced power consumption and cost, and shorter reconfiguration times since the partial bitstreams are smaller.

2 Motivation and Related Work

There exists a lot of work on the topic of partial reconfiguration: the authors of [7] explore the resulting reconfiguration time and bitstream size and simply provide evidence for the claims made in [6]. The paper [8] follows in a similar vein, but additionally proposes to utilize Block Random-Access Memory (RAM) to reconfigure regions even quicker but at the cost of increased resource utilization. In [1] the authors introduce a framework to automate aspects of the processes required to implement partial reconfiguration on Xilinx devices, a feature which is natively covered almost completely by the Xilinx Vivado tool nowadays. This means, partial reconfiguration can be used quite easily by hardware developers, but there is no simplified way to create a bitstream that supports it through HLS or PYNQ. The goal of this project is to create a framework that enables software developers to utilize dynamic partial reconfiguration and hardware acceleration through the simple and familiar Python programming language. As a proof of concept, an image processing pipeline with different regions and dynamic functions was implemented.

3 Partial Reconfiguration of Image Processing Pipelines

The framework consist of several components, the overall approach is depicted in Figure 3.

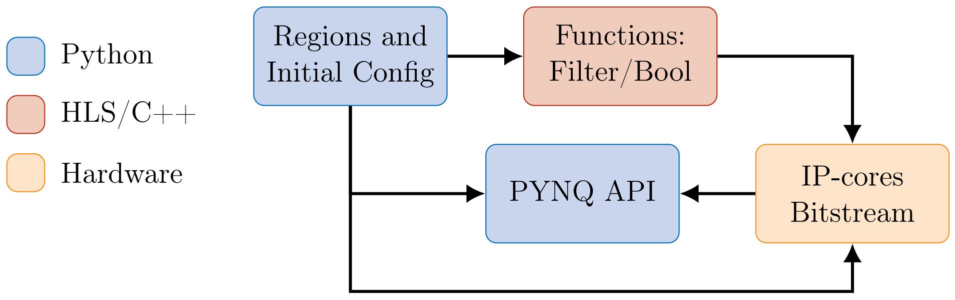


Figure 3: Framework components overview.

A Python script on the development PC holds information about the available functions and the initial configuration of all regions. All available functions were previously implemented using HLS and the Xilinx OpenCV library `xfOpenCV`, which allows for simple generation of hardware. Only the function interfaces need to be explicitly defined, everything else is analog to a software development process. The Python script calls the Vivado HLS compiler, which creates Intellectual Property (IP)-cores for all defined functions where the target clock frequency and target platform can be set. Afterwards, a base design for the selected platform is created with the Vivado tool in batch mode. Then the design is configured for dynamic partial reconfiguration and the regions and constraints are set accordingly. The resulting bitstreams and `.tcl` files can then be used to program the FPGA from within the PYNQ environment running on the SoC. After programming the FPGA, the individual accelerators can be utilized to increase the performance of the application.

3.1 Processing System

The prototype image processing pipeline that was implemented consists of the functions depicted in Figure 4.

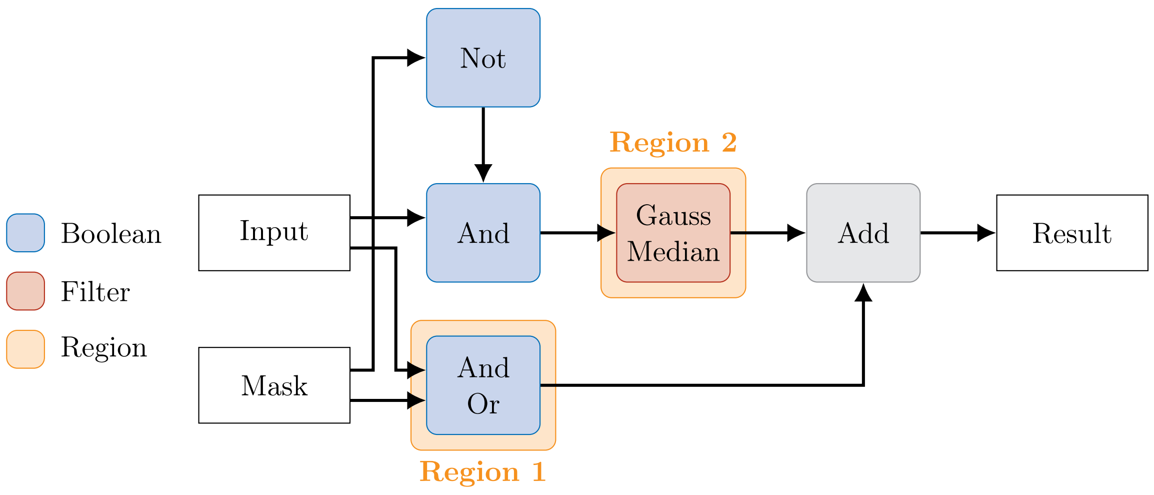


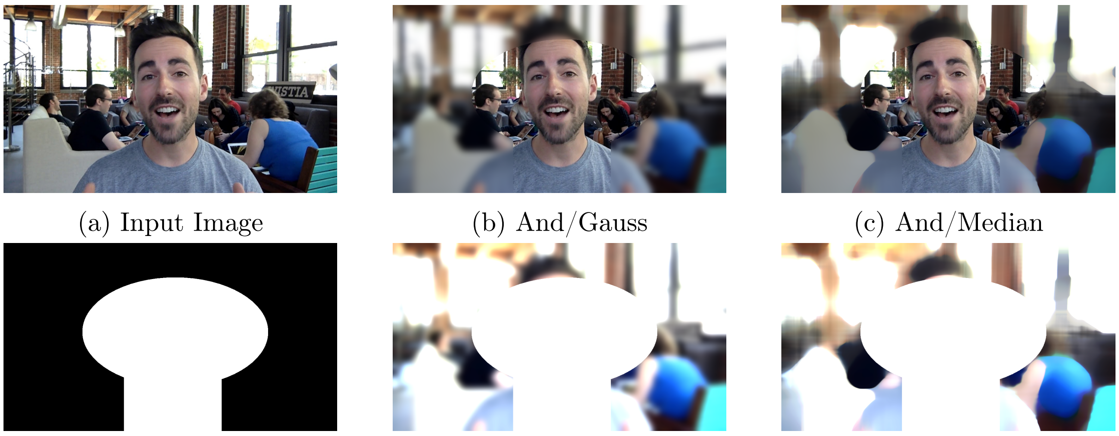
Figure 4: Implemented prototype algorithm.

While the pipeline consists of multiple functions, only the boolean functions “And” and “Or” and the filter functions “Gauss” and “Median” were implemented as dynamic partial regions with two different functions that can be inserted in each region respectively. The input and resulting images are shown in Figure 5.

To amplify the filter results, the kernel size was set to 81×81 but only to produce these showcase images. To obtain the performance results the kernel size was set to 3×3 for both the hardware and the software implementation. The characteristics of the different filters clearly show in the results. While the combination of the boolean “And” and both filters produces an image that could be used in a real world application, the “Or” operation does not produce any useful results and was only used to demonstrate the partial reconfiguration capabilities of the processing pipeline.

3.2 Programmable Logic

Three overlays were created throughout the course of this project:


(a) Input Image (b) And/Gauss (c) And/Median

(d) Mask (e) Or/Gauss (f) Or/Median

Figure 5: Inputs and results of the image processing pipeline.

- Base: this bitstream contains two of the four possible pipelines, namely the combination "And" and "Median", as well as the "Or" and "Gauss" pipeline. This serves as comparison for the design that implements the partial reconfiguration with regards to resource utilization.
- Streaming: similar to the base bitstream but only implements one pipeline. Instead of going through the PS RAM, the HDMI video streams are entirely processed on the PL. The PYNQ API is only used to start the processing and does not interact with the PL during runtime. This serves as another data point when comparing the resulting performance.
- Partial: this bitstream implements the algorithmic depiction of the pipeline as given in Figure 4. The entire design and all aspects of partial reconfiguration were automatically created by the Python and `.tcl` build scripts.

In the "Base" and "Partial" designs, the data transfers happen through Advanced eXtensible Interface (AXI) streaming interfaces that communicate with the RAM of the development board. To execute the pipelines and utilize the partial reconfiguration, the hardware has to be programmed with the static part of the partial bitstream and the desired functions for each region from within the PYNQ environment. Afterwards, the Python wrapper for an AXI Direct Memory Access (DMA) is used to send the images from the RAM to the acceleration cores and read the result back into memory.

4 Evaluation and Results

All designs were created using the 2021.1 version of the Vivado tool, since this is the first version that supports partial reconfiguration when working with a block design. Only one clock set to 100 MHz was used in the entire design. The measurements were conducted with full HD images with three data channels, which results in about 2 MB per image. To obtain results for the frames-per-second measurements, each filter was run 100 times and the resulting duration was used to calculate the metric.

4.1 Resource Utilization

Since hardware resources within a FPGA are limited, a more space-efficient design is able to achieve greater flexibility and function range. Table 1 gives an overview of the resource utilization percentage of the implemented overlays.

Overlay	Slice LUT [%]	BRAM [%]	DSP [%]
Base	54.52	9.29	35.45
Streaming	62.43	16.43	12.18
Partial	31.78	5.36	10.91

Table 1: Resource utilization percentage of the implemented overlays.

The "Base" overlay contains two parallel pipelines and thus uses more resources than the "Partial" overlay. This is expected but it also clearly shows the advantages of dynamic partial reconfiguration. Even with the increased resource utilization, the "Base" overlay is not able to fully replicate the functionality of the "Partial" overlay, the combinations "Or" and "Median", as well as "And" and "Gauss" are missing. This feature disparity will only increase with a growing number of regions and functions available in the block design with partial reconfiguration. The "Streaming" overlay uses even more resources since it also needs to accommodate various IP-cores for raw HDMI signal handling and processing. This overlay does not need to utilize the PS RAM and thus the performance is slightly better as the next section will show. Integrating partial reconfiguration into this design would also be possible, but was not done during this project due to time constraints. However, it would have been a good fit since the resources are already used quite heavily and adding further functionality not only increases the bitstream generation time but is also increasingly likely to fail since the synthesis tool is not able to create a hardware implementation that meets the required timings.

4.2 Performance

Another important aspect of creating a hardware/software co-design is the resulting performance of the system. Table 2 shows the latency and calculated frames-per-second metric of the PS and PL execution respectively.

All implementations execute the algorithm depicted in Figure 4 with the "And" and "Median" function selected, the "Base" overlay also computes the "Or" and "Median" pipeline but in parallel, so this should not add to the resulting execution time. Those measurements clearly show the advantages of hardware acceleration in this context, all hardware implementations are about 15 times faster than the execution in software. Unfortunately, all designs are not able to achieve a frame rate of 60 frames/s, but this

Overlay	Latency [ms]	Performance [frames/s]
Base	22.73	43.98
Streaming	20.83	48.00
Partial	23.35	42.82
Software	365.03	2.73

Table 2: Performance results of different implementations.

is expected behaviour since the amount of pixels in a full HD image is about 2 million which is equivalent to the amount of required clock cycles. This results in an optimal execution time of 20.736 ms, which the "Streaming" overlay is almost able to achieve. Some additional clock cycles are required to fill the intermediate First in, First out (FIFO) buffers and processing pipelines within the individual functions. Since the "Base" and "Partial" overlay use an AXI DMA to transfer the data from and to the main memory, some additional delays were introduced. Although the specification of the memory [3] indicates that it would support bandwidths of about 131.25 MB/s, which would result in approximately 66 frames/s, the additional delay of the AXI DMA limits this to the results obtained in Table 2.

5 Conclusion and Outlook

In this project an image processing framework prototype has been implemented. The framework is able to create computer vision pipelines with arbitrary building blocks after they were defined using HLS and connected as given in Figure 4. The entire overlay creation process is taken care of and software developers are able to utilize the hardware acceleration, resulting in an application that runs about 15 times faster than its software equivalent. With this partial reconfiguration approach, the resulting hardware design requires less resources and is able to accommodate more functionality than a regular hardware design.

In a future project, the framework could be adapted to implement the partial reconfiguration into the "Streaming" overlay, which would greatly benefit from additional functions that do not require further hardware resources since the required stream processing IP-cores already utilize many resources. Additionally, a Network-on-Chip (NoC) could be implemented that allows for reconfiguration of the partial region connections and thus more flexibility with regards to the parts of the algorithm that should be accelerated.

References

[1] Christian Beckhoff, Dirk Koch, and Jim Torresen. "Go ahead: A partial reconfiguration framework". In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines.* IEEE. 2012, pp. 37–44.

[2] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[3] Digilent Inc. *Pynq Z1 Specification.* URL: `https://reference.digilentinc.com/programmable-logic/pynq-z1/reference-manual?redirect=1` (visited on 08/19/2021).

[4] Tobias Kalb et al. "TULIPP: Towards ubiquitous low-power image processing platforms". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS).* IEEE. 2016, pp. 306–311.

[5] Lester Kalms and Diana Göhringer. "Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* IEEE. 2017, pp. 1–4.

[6] Cindy Kao. "Benefits of partial reconfiguration". In: *Xcell journal* 55 (2005), pp. 65–67.

[7] Wang Lie and Wu Feng-Yan. "Dynamic partial reconfiguration in FPGAs". In: *2009 Third International Symposium on Intelligent Information Technology Application.* Vol. 2. IEEE. 2009, pp. 445–448.

[8] Ming Liu et al. "Run-time partial reconfiguration speed investigation and architectural design space exploration". In: *2009 International Conference on Field Programmable Logic and Applications.* IEEE. 2009, pp. 498–502.

[9] Felix Winterstein, Samuel Bayliss, and George A Constantinides. "High-level synthesis of dynamic data structures: A case study using Vivado HLS". In: *2013 International Conference on Field-Programmable Technology (FPT).* IEEE. 2013, pp. 362–365.