# IMAGE PROCESSING ON PYNQ FPGA

# The project: Keypoint Descriptors

For low-level processing, FPGA devices are excellent candidates because they support data parallelism with high data throughput. Most previous keypoint descriptor formulations have inefficient FPGA implementations or deliver relatively poor information about the observed scene. In this work, I implanted a keypoint descriptor algorithm that aims for dense feature matching, due to time and scheduling I took the keypoint step from the overall CMFD process.

**So, how does this work?**

Keypoint descriptor algorithms will find some “interesting” keypoints(like nose, eyes) in the image by using edge/corner detection techniques and thresholding. So, we have stable keypoints that are scale-invariant and rotation invariant. The detection of key points in an image is nothing but selecting the points on the image which are good features and the descriptors are the representation of a point’s local neighborhood

A picture containing wall, indoor, several

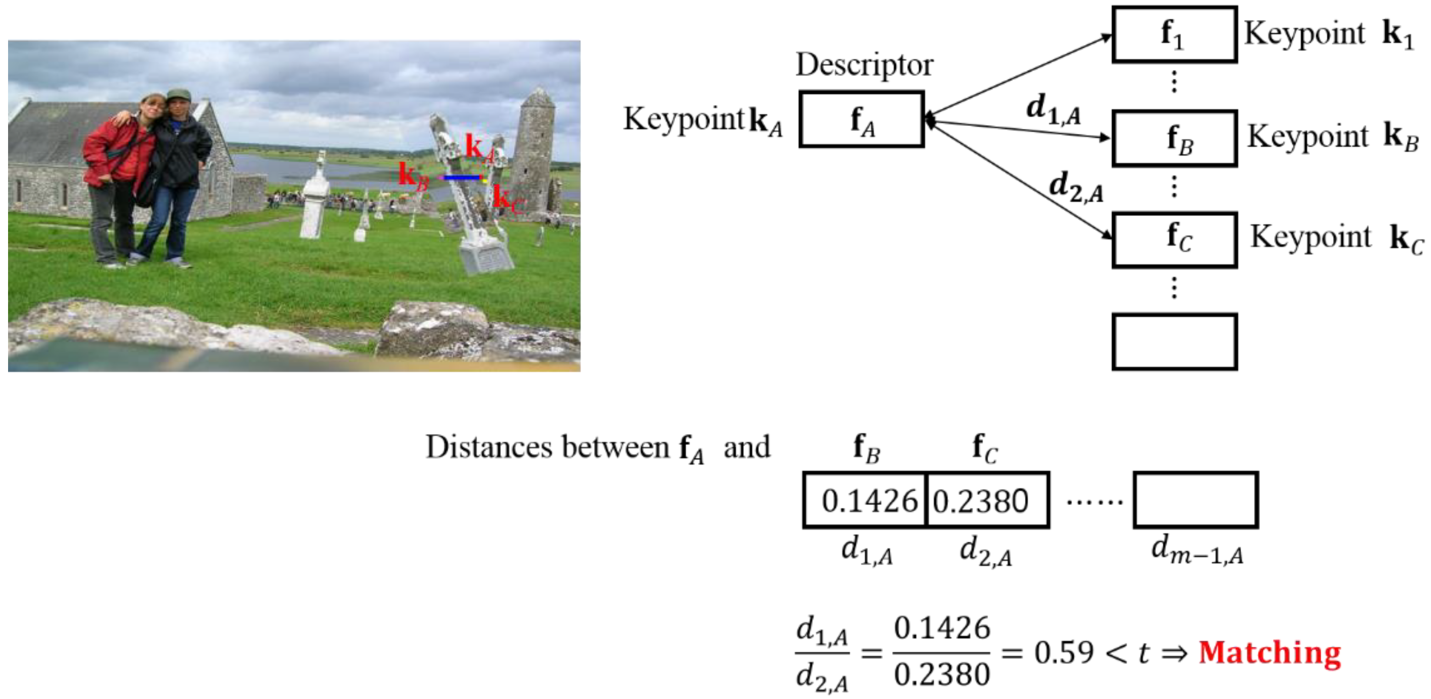
Description automatically generated

Figure 1 Example of correct keypoint matching using the proposed descriptor, Found 413 matches

# The FGPA

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing.

**System Overview**

**Graphical user interface, diagram, text, application, chat or text message

Description automatically generated**

Figure 2 the image flow is through AXI-stream interface from host to kernel back to host

**Vitis and Vivado HLS**

Vivado is the Hardware Development suite used to create a VHDL, Verilog, or any other HDL design on the latest Xilinx FPGA. In other words, when you need to translate your VHDL design into a configuration file to be downloaded into a Xilinx FPGA, you need Vivado framework.

Vivado is an integrated tool that allows you to perform the complete design flow for a Xilinx FPGA:

* Simulate
* Synthesize
* Map
* Route
* Analyze Timing
* Create a bit-stream FPGA configuration File
* Configure FPGA
* Debug the FPGA using ILA (Integrated Logic Analyzer)

**Vitis HLS**

In the Vitis application acceleration flow, the Vitis HLS tool automates much of the code modifications required to implement and optimize the C/C++ code in programmable logic and to achieve low latency and high throughput. The inference of required pragmas to produce the right interface for your function arguments and to pipeline loops and functions within your code is the foundation of Vitis HLS in the application acceleration flow. Our project following the Vitis HLS design flow:

* Compile, simulate, and debug the C/C++ algorithm.
* View reports to analyze and optimize the design.
* Synthesize the C algorithm into an RTL design.
* Verify the RTL implementation using RTL co-simulation.
* Package the RTL implementation into a compiled object file (.xo) extension, or export to an RTL IP.

**#pragma HLS interface**

Vitis HLS pragmas and directives let you configure the synthesis results for your code. HLS Pragmas are added to the source code to enable the optimization or change in the original source code. Every time the code is synthesized, it is implemented according to the specified pragmas.

**AXI4-Burst Mode (m\_axi)**

For fast speeds we use an Optimization Directive Interface called s\_axilite, or the set\_directive commands is associated with a bust mode solution. This is for faster computation results which are stored in a buffer and are written to global memory in a burst.

Graphical user interface

Description automatically generated

Figure 3 Opened Directive for the option to select Modify Directive

**Vitis Results**

**A screenshot of a computer

Description automatically generated**

Figure 4 Initial code with some basic $pragma

**A screenshot of a computer

Description automatically generated**

Figure 5 Optimization with compact streaming interface and pipelining the sub-functions

**Package the RTL implementation export to an RTL IP**

After the verifying, Synthesizing and running the Simulation, the final step in the Vitis HLS flow is to export the RTL design. Click the Export RTL command in the Flow Navigator to open the Export RTL dialog box shown in the following figure.

Graphical user interface

Description automatically generated

Figure 6 Export RTL dialog box

**Vivado: Generating bitstream from RTL code**

We need to Import RTL code and can be found on the imported RTL for folder here

Graphical user interface, application, chat or text message

Description automatically generated

Figure 7 IP location from the Vitis RTL

**Created Design**

After the IP import and adding a couple of interconnects and DMA this IP is produced

**Graphical user interface, diagram

Description automatically generated with medium confidence**

Figure 8 Created Block Diagram from IP Integrator

**Generate Bitstream**

In Sources, right click on design\_1 and select Create HDL Wrapper

**Bitstream, .hwh, and addresses**

To locate the needed .hwh and bitstream go to sources . . expand design\_1\_wrapper::design\_1\_i::design\_1::axi4\_sqrt\_0::design\_1\_axi4\_sqrt\_0\_0::inst : axi4\_sqrt, double click on axi4\_sqrt\_sqrt\_s\_axi\_U , and note the address for ipHCD write as 0x10 , 0x18 and 0x00 respectively. We need this address in our host program

**Graphical user interface, application, Word

Description automatically generated**

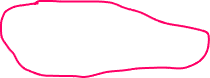


Figure 9 Under Sources, open mul\_test\_mul\_io\_s\_axi.v, scroll down and note addresses for in and out ports

**Host Program**

We implement a host program in python (host), which can be run by using Jupyter Notebook. To interact with the IP first we need to load the overlay containing the IP b the following lines of code

# designate a bitstream to be flashed to the FPGA

ol = Overlay("/home/xilinx/jupyter\_notebooks/CapstoneF/design\_1.bit")

ol.download() # flash the FPGA

**What is the difference between an ASIC and an FPGA?**

ASIC and FPGAs have different value propositions, and they must be carefully evaluated before choosing any one over the other. Information abounds that compares the two technologies. While FPGAs used to be selected for lower speed/complexity/volume designs in the past, today’s FPGAs easily push the 500 MHz performance barrier. With unprecedented logic density increases and a host of other features, such as embedded processors, DSP blocks, clocking, and high-speed serial at ever lower price points, FPGAs are a compelling proposition for almost any type of design.

**Results/Comparison**

****

Figure 10 Estimated CPU vs FPGA results

# ADALM-PLUTO Radio Communications for PYNQ-Z2

To connect the pluto with the ability to transmit an image, I tried to use GNU Radio which seems to be a popular way with a lot of control using the blocks. Was unable to run GNU Radio due to flaky PYNQ VNC. In the end able to install SDR drivers directly on to the PYNQ

* libiio, Analog Device’s “cross-platform” library for interfacing hardware
* libad9361-iio, AD9361 is the specific RF chip inside the PlutoSDR
* pyadi-iio, the Pluto’s Python API, this is our end goal, but it depends on the previous two libraries

then able to connect using the following lines of code

python3

import adi

sdr = adi.Pluto('ip:192.168.2.1') *# or whatever your Pluto's IP is*

sdr.sample\_rate = int(2.5e6)

sdr.rx()

Chart

Description automatically generatedChart

Description automatically generated

Figure 11 loopback results

**Future Improvements**

* There are number of approaches used to detect and matching of features as SIFT, SURF (Speeded up Robust Feature), FAST, HCD etc. SURF has become a more useful approaches to detect and matching of features because of it is invariant to scale, rotate, translation, illumination, and blur.
* Have all CMFD steps run on the FPGA
* Use GNU Radio for image transmission and receive
* Create a smaller overall device footprint