HILO: High-level and Low-level Co-design, Evaluation and Acceleration of Feature Extraction for Visual-SLAM using PYNQ Z1 Board

Muhammad Bilal Akram Dastagir[1], **Omer Tariq**[1] **and Dongsoo Han**[1]
[1]School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

Abstract
Image features are widely employed in embedded computer vision applications, from object identification and tracking to motion estimates, 3D reconstruction, and visual simultaneous localization and mapping (VSLAM) applications. Due to the real-time needs of such applications over a continual stream of input data, efficient feature extraction and description is critical. Significant-speed processing is often associated with high power consumption, yet embedded systems are mostly power and resource-limited, making the development of power-aware and compact solutions all the more important. The performance of the low-cost feature detection and description algorithms implemented on particular embedded devices is evaluated in this work (embedded processing units, GPUs, and FPGAs). We implemented the ORB-based feature extraction hardware accelerator using PYNQ overlay in the embedded PYNQ Z1 board. We demonstrated that a speedup of 8.38x was achieved utilizing a hardware-accelerated core compared to the algorithm running on a processor-based software solution.

Keywords
Feature Extractor, Feature Detector, Visual SLAM, FPGA, Acceleration, PYNQ, Overlay, ORB-SLAM

1. Introduction

Simultaneous Localization and Mapping (SLAM) has received much recognition in recent years due to its path planning, map building, and navigation, which offers important technical advantages in Autonomous Navigation Systems [1] [2] [3]. Simultaneous localization and mapping, or SLAM, is a technique used to map an unknown location while simultaneously localizing the agent using SLAM on the same map. Visual SLAM (which uses pictures from cameras and other image sensors) and LiDAR SLAM are the two types of SLAM currently in use (which use a laser or a distance sensor). It can calculate the pose estimation and a 3D reconstruction of the area in various configurations, from hand-held sequences to cars being driven over several city blocks. Many researchers and scientists have been exploring and adapting the implementation of SLAM using cameras as Visual-SLAM due to its simplicity, cost-effectiveness, and rapid deployment compared to the alternatives sensors in embedded systems [4] [5]. Visual-SLAM has various applications, from autonomous driving cars to other domains

IPIN 2022 WiP Proceedings, September 5 - 7, 2022, Beijing, China
EMAIL: bilal@kaist.ac.kr (Muhammad Bilal Akram Dastagir); omertariq@kaist.ac.kr (Omer Tariq); ddsshhan@kaist.ac.kr (Dongsoo Han)
ORCID: 0000-0003-2990-4604 (Muhammad Bilal Akram Dastagir); 0000-0002-1771-6166 (Omer Tariq); 0000-0002-2396-1424 (Dongsoo Han)

like airborne navigation, Advanced Driver Assistance Systems (ADAS), Augmented Reality (AR), and Virtual Reality (VR), and has also been explored as a new emerging positioning and navigation solution [6] [7] [8]. One of the most noticeable low-level features insensitive to scale, rotation, noise, and light is the Scale-Invariant Feature Transform (SIFT) [9]. Like SIFT, the suggested Speeded-UP Robust Feature (SURF) [10] has a reduced computing complexity and performs almost as well. The Histogram of Gradients is used by both of them to find repeating keypoints in scale space and to produce descriptions (HoG). Despite SIFT/SURF obtaining high-quality features, real-time implementation of those features is challenging owing to heavy computation and memory access. Likewise, Visual-SLAM algorithms like feature-based, direct, or RGB-D camera-based approaches are very compute-intensive and thus cause a high latency in dealing with image processing algorithms, making it challenging on resource-constraint devices for real-time implementation on ADAS [11] [12] [13]. The literature only contains a relatively small number of entire embedded operational chains that adhere to the ADAS requirements for fast processing times, low power consumption, and area-aware design footprints. This problem is partially caused by the modern image processing algorithms' inherent complexity and the amount of data (number of pixels) that must be handled. One way to address this high performance computing issue is; either utilize Graphing Processing Unit (GPU) or Field Programmable Array (FPGA) for fast execution and real-time implementation of the algorithms, and there have been various research explored in this domain [14] [15]. Although the conventional FPGA design environment supports the algorithm acceleration but lacks software-like programming functionality, thus results in time-consuming application development [16].

This paper presents an efficient high-level and low-level co-design, evaluation and acceleration of real-time ORB feature extraction for visual-SLAM using the PYNQ Z1 board. We evaluate the implementation of feature detection using the PYNQ platform, which has system-on-chip (SoC) encapsulation of the Arm-processor and FPGA by a python-based development environment and tools. This integrated hardware/software co-design environment encapsulating the FPGA formed a robust, rapid, and reliable co-framework in which execution of the compute-intensive algorithms increased significantly.

The rest of this paper is organized as follows. Section II is dedicated to the literature review of the visual SLAM in software and hardware domains. Section III reviews the background information of PYNQ architecture and its co-design with the ORB-based feature algorithm. Section IV discusses the evaluation of our experimental setup for the proposed hardware feature extractor, the implementation using hardware-software co-design, and the promising results. In Section V, we summarize our conclusion and future work.

2. Related Work

The simultaneous localization and mapping (SLAM) problem has now received considerable attention from the robotics world, and many scholarly solutions have been put forth. Lee et al. [17] suggested a real-time RGB-D 3D SLAM system based on a GPU and only an RGB-D sensor. The 3D SLAM system's operation speed is accelerated by GPU computing, with an average processing rate above 20 Hz. Giubilato et al. [18] proposed a ROS interfaced visual SLAM set in an indoor setting using a 6-wheeled ground rover equipped with a stereo camera, LiDAR, and TX2 computer platform. By leveraging an embedded computer platform to report realistic results during real-life mobile robot operations, the investigation demonstrated image rectification on the CPU and GPU and presented a fresh benchmark for visual SLAM algorithms. Peng et al. [19] provided a comprehensive and quantitative performance assessment of ORB-SLAM2 and OpenVSLAM on the platform. Tianji, et al. [20] investigated the front end of visual SLAM using an integrated GPU and developed a front-end solution for parallelization. The visual SLAM system was deployed on the embedded platform using GPU parallelization, and the EuRoC MAV dataset was used to verify the system's efficacy.

On the other hand, D. T. Tertei et al. [21] proposed an efficient architecture for tri-matrix hardware accelerator based on Virtex5 XC5VFX70T FPGA that leverages matrix multiplications and updates the

cross-covariance matrix in the correction loop of a 3D EKF-based SLAM algorithm. The accelerator is executed at 44.39Hz and permitted a real-time visual SLAM with 45 observed and corrected landmarks. Vourvoulakis et al. [22] demonstrated a completely pipelined and optimized architecture for detecting SIFT keypoints and extracting SIFT descriptors in real-time. The system is designed for robotic vision applications and runs on a Cyclone IV FPGA chip with a clock speed of 21.7 MHz and a feature extraction time of 46 nanoseconds. Furthermore, the suggested solution has good responsiveness and repeatability values, and its matching ability is directly similar to SIFT implementations based on floating-point software. Fang et al. [23] proposed an ORB-based feature extractor capable of running at 203 MHz to reduce energy consumption and computational intensity and outperformed several state-of-the-art processors in terms of latency, energy, and performance. Qi Ni et al. [24] introduced SURF feature detection, BRIEF descriptor construction, and matching algorithms for binocular vision systems that provided feature point correspondences and parallax information at 162fps @640 480 on a ZYNQ SoC device. Ayoub Mamri et al. [25] proposed implementing the ORB-SLAM2 algorithm targeting a heterogeneous hardware/software optimization approach. They executed the optimization in an FPGA-based heterogeneous embedded architecture and compared the outcomes with those of other heterogeneous architectures, including powerful embedded GPUs (NVIDIA Tegra TX1) and high-end GPUs (NVIDIA GeForce 920MX). Their implementation utilized high-level synthesis-based OpenCL for FPGA and CUDA for NVIDIA targeted devices. T. Imsaengsuk et al. [26] proposed a feature detector and descriptor based on the ORB algorithm that could be accelerated by exploiting parallelism and pipelining. They applied a binomial filter instead of a Gaussian filter, used a heap sorting algorithm to balance the number of feature points and designed the data processing pipeline and orientation estimation method to design optimal hardware architecture. In the proposed design, they maintained the consistency of corner numbers; however, their feature detector could only detect single corner points.

3. HILO co-design using PYNQ overlay

3.1. PYNQ

PYNQ [27] [28] is a framework created by Xilinx to make it easier for software programmers to use FPGAs. The PYNQ package allows designers to utilize hardware designs from Python instead of C-based drivers. For instance, designers may conduct DMA transfer with just a few lines of Python code. Through Jupyter notebooks, designers may leverage FPGA in the PYNQ framework. The system-on-chip zc7020 integrates an FPGA and a dual-core CPU onto a single chip, and the PYNQ overlay is installed on the PYNQ-Z1 FPGA board.

3.2. OpenCV

The widely used Open Source Computer Vision Library (OpenCV) is the foundation for various image processing applications. It is open-source and contains over 2500 optimized computer vision and machine learning algorithms. The library is written in C++ and includes a Python API, making it ideal for the PYNQ platform. OpenCV has much support for image and video file handling and camera interfacing. If the platform supports vector units like SSE or NEON, OpenCV functions use them. CUDA and OpenCL interfaces are being actively developed to support GPU execution. The PYNQ software already includes OpenCV, so the functions can be used in a Python application because of the library's benefits.

3.3. Hardware Library

Figure 1 depicts the system architecture, including the layers and their connections. PYNQ provides a web-based interface for programming Python applications using notebooks. On a single board, several of these notebooks can run in parallel. PYNQ provides modules to interface with programmable logic and the standard Python libraries. Modules such as Direct Memory Access (DMA) and Memory Mapped I/O (MMIO) provide different ways to exchange data with the hardware IP cores. The Overlay module

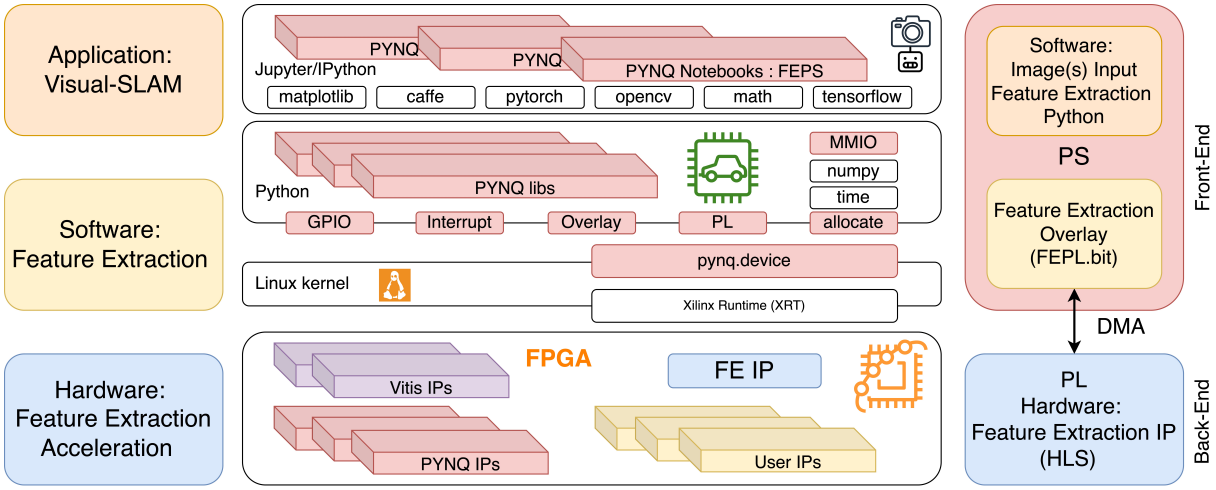

Figure 1. HILO co-design framework using PYNQ

controls the bit file loaded into the programmable logic and the hardware hierarchy. When using the DMA data exchange, physically contiguous memory must be allocated for the DMA IP-core to access a data block in ascending order. As a result, the link module provides contiguous memory allocation in a unique region of the Linux memory hierarchy.

The introduced library is divided into two sections. The kernels for feature extraction are included in a custom bit file with the library. The second component is the Python library, which provides the application API and handles the correct interaction with the hardware. It accesses the programmable logic through the PYNQ framework's interfaces. This design allows for dynamic target computing unit selection and future expansions. These operations can be efficiently implemented using a sliding window operation on FPGAs. This method allows for streaming by caching image lines. The filter mask's height determines the number of cached lines.

3.4. Overlay

The OpenCV library of the main PYNQ project does not support image processing accelerators running on FPGAs. Every OpenCV function call on the PYNQ platform is executed on the ARM CPUs with their NEON units. Therefore, we decided to make the first step towards supporting FPGAs and developing a python library that extends the capabilities of OpenCV and sources computation-intensive tasks out to the programmable logic. This library concept consists of several parts. The hardware design for the programmable logic part of the PYNQ platform is called an overlay [29]. Additionally, a Python library interfacing the overlay and providing the API to the user is needed.

In Fig 2, PYNQ based notebook frame has been shown how both hardware and software libraries are included in the python framework for rapid prototyping and development of the application.

Similarly, in Fig 3, it is demonstrated that how FPGA bit stream file is being loaded along with the initialization and assignment of DMA AXI with python variables for hardware-software handshaking for the acceleration.

3.5. ORB Feature Extractor

A process for extracting features from an image is known as feature extraction. It is divided into two parts: (Oriented Feature from Accelerated Segment Test) based feature extraction and BRIEF (Binary Robust Independent Elementary Features) based feature descriptors computation, with Gaussian Filter processing available only in the FPGA environment, which allows parallel computation, unlike software implementation.

```
#Importing Libraries Python and FPGA Related
from pynq import Overlay
from pynq import Xlnk
import matplotlib.pyplot as plt
import numpy as np
import cv2
from time import time
import time as tm
import cv2
```

Figure 2. Importing Software, Hardware Libraries using Overlay

```
#Loading FPGA Drivers

start = tm.time()
xlnk = Xlnk()
overlay       = Overlay('FEPL.bit') #Loading FPGA Bit Stream File into Jupyter w
ith Overlay
dmaFilter     = overlay.PS2PL_Filter_AXI_DMA     # for Sending to FPGA
dmaExtractor  = overlay.PS2PL_Extractor_AXI_DMA  # for Sending to FPGA
dmaDescriptor = overlay.PL2PS_Descriptor_AXI_DMA # for receiving from FPGA
stop = tm.time()
executionTime = (stop - start)
print("FPGA Drivers Loaded Successfully with Execution Time = ", executionTime,
" Seconds")
```

```
FPGA Drivers Loaded Successfully with Execution Time =  1.695509910583496  Seco
nds
```

Figure 3. Loading FPGA bit File using Overlay into Jupyter PYNQ

3.5.1. Gaussian Kernel The image is blurred using a Gaussian filter to decrease noise and eliminate speckles from the image. It is critical to filter out high-frequency components not related to the gradient filter in use; otherwise, it may lead to false edge detection.

The Gaussian function is represented in Equation 1 as

$$G(x) = \frac{1}{\sqrt{2\,\pi\,\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

Where σ is the distribution's standard deviation The mean of the distribution is assumed to be 0 [30].

The Gaussian function is a smoothing operator that provides a probability distribution for noise or data. It is employed in a variety of study fields. While working with images, we will use the two-dimensional Gaussian function, the product of two 1D Gaussian functions. The Gaussian filter employs the 2D Gaussian distribution function as a point-spread function, which is accomplished by the 2D Gaussian distribution function with the image to obtain a discrete approximation to the Gaussian function.

3.5.2. Extractor To implement the real-time implementation of a SLAM using a mobile robot with limited computational resources, features from accelerated segment test (FAST) are an excellent choice in the category of corner detection algorithms comparable to SIFT and SURF, which could be utilized in the feature extraction of points and map the objects in image processing tasks. To calculate the orientation component of FAST, ORB employs the intensity centroid. The intensity centroid may infer an orientation that a corner's intensity skewed from its central point. The image moment may be calculated as Eq. 2:

$$m_{pq} = \sum_{x,\,y} x^p y^q I\,(x,\,y) \tag{2}$$

In equation (2), p and q are natural numbers representing the moment order in each dimension, and I (x, y) represents the strength of the pixel at the relative location. It is feasible to calculate the orientation centroid using this definition:

$$C = \left(\frac{m_{01}}{m_{00}}, \frac{m_{10}}{m_{00}}\right) \tag{3}$$

then calculate the centroid C, OC, and the corner's center point O using Eq. 4.

$$\phi = a\tan 2(m_{01}, m_{10}) \tag{4}$$

The quadrant-aware arctangent is known as atan2. The possibility to compute sin(Ø) and cos(Ø) using the m values in the following Eq. 5:

$$\sin(\phi) = \frac{m_{10}}{\sqrt{m_{01}^2 + m_{10}^2}}, \cos(\phi) = \frac{m_{10}}{\sqrt{m_{01}^2 + m_{10}^2}} \tag{5}$$

A pixel's status as a corner is determined by the segment test, and oFAST is made up of the placement calculation. To create a set of features, this data is used.

The pseudo-code for the FAST algorithm [31] is explained as:

Input: Any RGB or greyscale Image

Output: A set of key points for selected image

(i) Initiate FAST object "FastFeatureDetector_create()" with default values

(ii) Find the key points using the "detect()" method.

(iii) Draw the key points using the "drawKeypoints()" method.

(iv) Disable the Non-Max Suppression (NMX) to remove the bounding box.

(v) Display the image with the key points that appeared

3.5.3. Descriptor We will use the rotation-aware Binary Robust Independent Elementary Features (rBRIEF) algorithm as a descriptor block. The brief algorithm's major points are all converted into a binary feature vector via the BRIEF technique, which can subsequently be used to represent an object. A feature vector containing only the integers 1 and 0 is considered binary. All in all, a feature vector, a 128–512 bit string, is used to represent it.

Starting with image smoothing with a Gaussian kernel, we avoid the descriptor being sensitive to high-frequency noise. Select a random pair of pixels in the area around that important point that is defined. A pixel's specified neighborhood is represented by a square of defined width and height, known as a patch. The first pixel in each random pair is chosen from a Gaussian distribution with a stranded deviation, also known as a spread of sigma, centered on the important point. In a random pair of pixels, the second pixel is chosen randomly from a Gaussian distribution with a standard deviation or spread of sigma by two, centered on the first pixel. The appropriate bit is either set to 1 or 0. Once more, choose a random pair and give them a value. For a crucial location in a 128-bit vector, briefly repeat this procedure 128 times. Build a vector that looks like this for each image. Moreover, BRIEF is rotation invariant; therefore, ORB utilizes rotation-aware BRIEF [32] [33]. The in-plane rotation invariance of ORB, one of its most appealing characteristics, is achieved by rotating the binary test coordinates of rBRIEF in accordance with the orientation discovered earlier by oFAST.

A set of binary intensity tests is used to create the rotation-aware BRIEF (rBRIEF) descriptor. A binary test τ is defined as follows:

$$\tau(p_1, p_2) = \begin{cases} 0 & , I(p_1) < I(p_2) \\ 1 & , I(p_1) \geq I(p_2) \end{cases} \tag{6}$$

where I(p_i) represents the intensity of the point p_i, and p_1 and p_2 represents the 2D points. A vector of n binary tests is used to define the feature:

$$f_n(p) = \sum_{1 \leq i \leq n} 2^{i-1} \tau(p_{1_i}, p_{2_i}) \tag{7}$$

Before carrying out these tests, it is recommended to smooth the image first, maybe using a Gaussian filter. The vector normally has a dimension of 256.

The algorithm for Rotation-aware BRIEF (rBRIEF) [34] is explained as:

(i) For all of the training patches, compare each test.

(ii) By arranging the tests in order of their deviation from a mean of 0.5, the vector T is generated.

(iii) The first test should be taken out of T and put in the result vector of R, applying greedy search.

(iv) Take the next test from T and contrast it with the other tests from R. The absolute correlation is ignored if it exceeds a certain threshold; otherwise, it is added to R.

(v) Until R has 256 tests, repeat the preceding procedure. Increase the criterion and try again if the number of outcomes is fewer than 256.

3.6. Hardware accelerated ORB feature extractor

The proposed FPGA based hardware accelerator for the feature extraction system is shown in Figure 4. ARM multi-core processors handle image loading and other computations in the system. Connected memory systems, feature extraction accelerators, and ARM processors all use the AXI bus. At the top system-level, the feature extraction accelerator is a data accessing part and a kernel part. Instruction memory, control unit, and feature extractor make up the kernel. Input buffer, output buffer, and DMA interface make up the data accessing section. The DMA interface uses the AXI bus to directly access the image's data stream and stores it in the feature extractor's input buffer. The feature extractor's results are saved in the output buffer before being sent to the ARM multi-core System, which uses them for further processing.

It is necessary to use hardware elements that support a sliding window access pattern while processing an input stream of pixels from onboard memory or image sensors. This arrangement efficiently takes advantage of the spatial and temporal localization of the processes. The accelerator receives the pixel stream at a rate of one pixel per cycle, which is the standard order for image data. For feature detection and rBRIEF descriptor computation, the pixel stream travels two cooperative data routes. A functional component of the first route is the FAST feature detector. Blocks for orientation computation, BRIEF descriptor generation, and Gaussian image smoothing make up the second route. The rBRIEF block design in this study blends replication with a static approach of pattern reordering. The pixel stream is delayed using the FIFO to keep the two paths in synchronization. After a specific number of fixed rotations, the FIFO structure allows components to leave the structure in FIFO order. A circular index indicating the place from which to read and write can be used to implement it in memory efficiently. The accelerator synchronizes all sliding windows required for properly operating the aforementioned functional blocks while only accessing each pixel of input images once. The utilization of tiling in the proposed architecture, which increases performance and flexibility to process various image resolutions while requiring less on-chip memory storage, is another crucial component. An image is divided into several little rectangular "tiles." using a tiling method. The accelerator can process each of those fixed-size tiles transparently since it makes no difference whether the incoming data is a fragment of a single image or a subset of a larger one. The accelerator's tiling enhances locality and lowers the need for on-chip memory.

The most difficult aspect of the ORB extraction is implemented in the BRIEF block since it needs to grant memory access to 512 places that rely on the feature angle to process the descriptor block. Finding a suitable schedule for these accesses is challenging since the angle relies on the input data. The tests' necessary data are stored in the implementation's 37×37 sliding windows. This window is synced with the other blocks to hold a patch centered on the feature candidate. The structure must permit the generation of the descriptor using the sliding window dataflow approach and random access to 512 places.

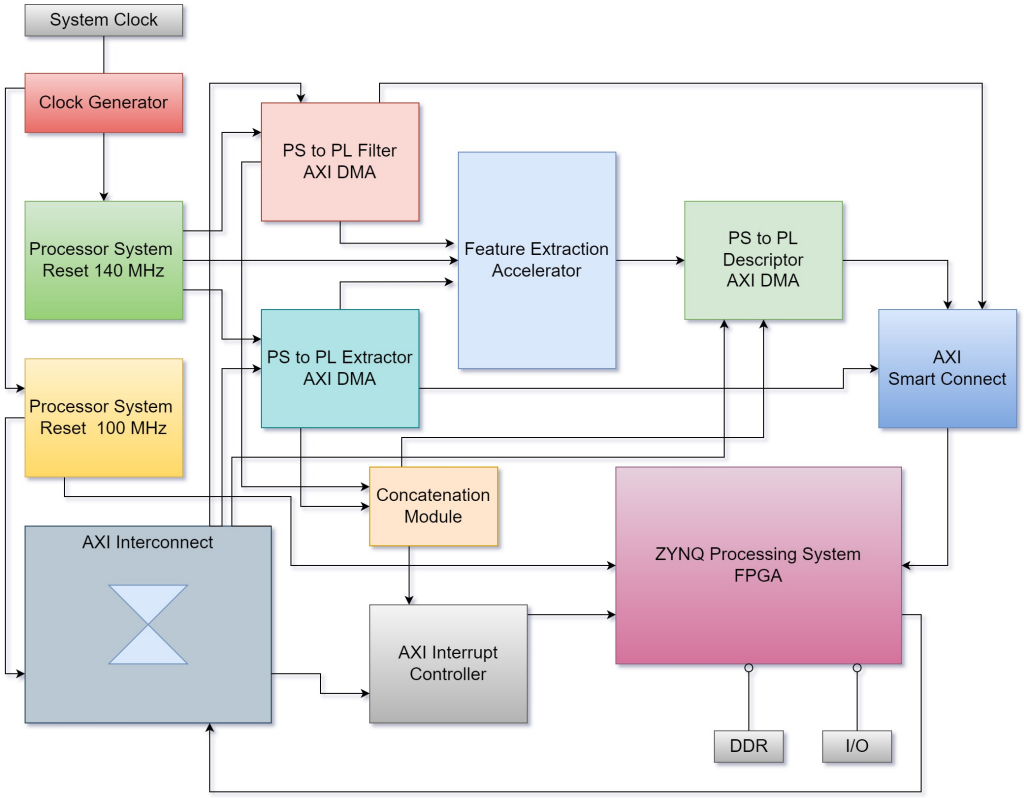

Figure 4. Proposed hardware accelerated ORB feature extractor

4. Evaluation

We study and compare the performance impacts of employing PYNQ for feature extraction application development using C, Python, OpenCV libraries, and custom accelerators. The various testing configurations used in the experimental setting are explained before the study and analysis of the results.

4.1. Experimental Setup

The Xilinx PYNQ platform [35] was employed for this study, which contains the Zynq system-on-chip component and DDR3 memory of 512 MB. The dual-core ARM CPU operates at 667 MHz, with the FPGA logic core operating at 125 MHz. Feature Extraction was performed on a 640x480 grayscale image in the experiment, which is a common step, and ORB was chosen as an accelerator because of the robust and open source libraries of image processing in visual SLAM [36].

Three alternative software and hardware setups have been utilized in the experimental analysis. C and Python performance must be compared to achieve our goal in an embedded development environment for feature extraction. A specially customized and incorporated OpenCV kernel was utilized in this experiment. Utilizing these libraries, the register transfer level intellectual property (IP) blocks for Gaussian filtering, the FAST Algorithm, and a rBRIEF detector were created and synthesized in the FPGA for production of the bitstream file to be utilized by the PYNQ ARM processor in the Juptyer Notebook employing overlay using dynamic memory allocation (DMA).

The algorithm of the feature extraction accelerator was developed using Vitis HLS tool employing rapid productivity. Three distinct streaming IPs have been developed: Gaussian Filter, Extractor, and Descriptor. The hardware-accelerated version on the FPGA fabric uses three IPs operating at 125 MHz. Multiple research articles prove that the feature extraction on FPGAs may outperform software implementations on processors and GPUs.

4.2. Implementation

```
def main():
    print("Program Started Successfully....!!!")
    imageName = ('KAIST_64.jpeg')


    runTimePL = runFeatureExtractionOnFPGA(imageName)
    runTimePS = runFeatureExtractionOnProcessor(imageName)
    print("The Total Speed-Up (Acceleration) is ", ((runTimePS/runTimePL)*
(1)),'x')
    print("The Total Improvement using FPGA  is ", (((runTimePS/runTimePL))*(10
0)),'%')

    print("Program Ended Successfully. ....!!!")
```

Figure 5. Main function of the processor and FPGA Functions

Figure 5, shows the primary function written in the python, which implements both software and hardware version of the feature extraction and test image. In addition, Fig 6 and Fig 7 show feature extraction implementation on the processor and FPGA, respectively.

```
def runFeatureExtractionOnProcessor(imageName):
    image = cv2.imread(imageName)
    orb = cv2.ORB_create(edgeThreshold=15, patchSize=31, nlevels=8, fastThreshol
d=20, scaleFactor=1.2, WTA_K=2,scoreType=cv2.ORB_FAST_SCORE, firstLevel=0, nfeat
ures=3989)
    start = tm.time()
    keyPoint = orb.detect(image)
    stop = tm.time()
    executionTime = (stop - start)
    print("The Feature Extraction(Detection) Execution Time (PS) = ", executionT
ime, " Seconds")
    print('Feature points are detected')
    print(len(keyPoint))
    imageKeyPoints = cv2.drawKeypoints(image, keyPoint, None, color=(0,255,0),
            flags=cv2.DrawMatchesFlags_DEFAULT)
    plt.figure(figsize=(10,7.5))
    plt.imshow(imageKeyPoints)
    plt.show()
    return executionTime
```

Figure 6. Feature Extraction implemented on Processor

4.3. Results and Analysis

Fig 8 shows the results of implementing the co-design of the feature extraction both on the PYNQ ARM Processor and FPGA and the Intel CPU implementation. The red marker depicts the features using FPGA, while green depicts using the processor. Similarly, blue depicts the OpenCV-python-based features on Intel CPU with 16Gb of DDR4 RAM. The timing information is presented in Table 1.

The results of executing Feature Extraction on three distinct hardware and software setups are shown in Fig 9, Fig 10 and Table 1: Python's performance on the Intel CPU is demonstrated. On the PYNQ

ARM A9 cores, a Python OpenCV implementation was achieved with minimal effort. Python was in combination with a hardware-accelerated core.The intricacy of portability and the need for cross-compilers and device drivers are decreased when using a platform like PYNQ, where Python serves as the primary interface for programmers to the hardware. Programming that reads and writes data via MMIO and DMA is made possible by PYNQ, which greatly reduces system design complexity. A developer may quickly construct, test, and modify their program using the profiling and debugging capabilities built into Python or available through libraries and packages. A hardware-accelerated core achieved a

```
def runFeatureExtractionOnFPGA(imageName):
    #Image Reading
    imageOriginal = cv2.imread(imageName)
    imageGray = cv2.cvtColor(imageOriginal,cv2.COLOR_BGR2GRAY)
    #Tx and Rx Buffer for AXI DMA
    txBuffer_AXI_DMA = xlnk.cma_array(shape=(640*480,), dtype=np.uint8)
    view=np.frombuffer(txBuffer_AXI_DMA,dtype = np.uint8,count = -1)
    np.copyto(view,imageGray.ravel(),casting='same_kind')
    rxBuffer_AXI_DMA = xlnk.cma_array(shape=(8192,10), dtype=np.int32)

    start = tm.time()
    dmaDescriptor.recvchannel.transfer(rxBuffer_AXI_DMA)
    dmaExtractor.sendchannel.transfer(txBuffer_AXI_DMA)
    dmaFilter .sendchannel.transfer(txBuffer_AXI_DMA)
    dmaFilter.sendchannel.wait()
    dmaExtractor.sendchannel.wait()
    dmaDescriptor.recvchannel.wait()
    rxBuffer_AXI_DMA.flush()
    bytes_read=dmaDescriptor.mmio.read(0x58)
    featurePointsNum = int(bytes_read/40) - 1
    stop = tm.time()
    executionTime = (stop - start)
    print("The Feature Extraction(Detection) Execution Time (PL) = ", executionT
ime, " Seconds")

    print(str(featurePointsNum)+' feature points are detected')

    for i in range(2):
        posX = rxBuffer_AXI_DMA[i][8]
        posY = rxBuffer_AXI_DMA[i][9]
        print('Descriptor: ')
        print(rxBuffer_AXI_DMA[i][0:7])
        print('\npositon:')
        print('x:'+str(posX)+'  y:'+str(posY)+'\n')

    #Drawing Features
    points = rxBuffer_AXI_DMA[0:(featurePointsNum-1)]
    pos = (points[:,8:10]).copy()
    posList=pos.tolist()
    for i in posList:
        cv2.circle(imageOriginal,(i[0],i[1]),2,(0, 0, 213),-1)

    imageDisplay = cv2.cvtColor(imageOriginal,cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(10,7.5))
    plt.imshow(imageDisplay)

    txBuffer_AXI_DMA.freebuffer()
    rxBuffer_AXI_DMA.freebuffer()
    return executionTime
```

Figure 7. Feature Extraction implemented on FPGA

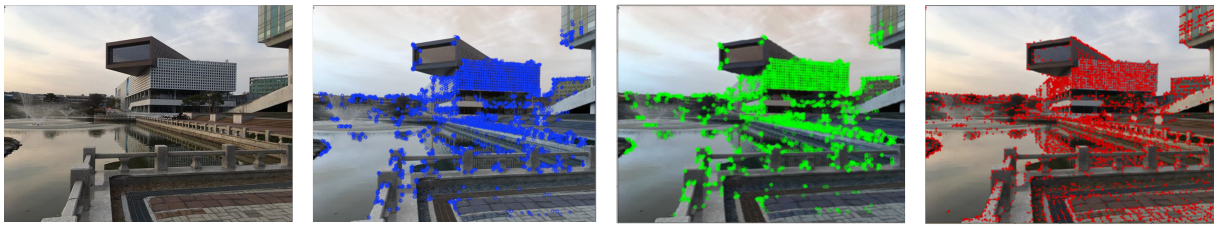

Figure 8. Feature Extraction using three configurations

Table 1. Experimental Result Comparison for Feature Extraction Execution Time

Configuration	Clock Frequecny	Time (s)	Speedup (x)
Intel CPU with 16Gb of DDR4 RAM	2.3 GHz 8-Core Intel Core i9	0.04291	2.43x Slower than FPGA
PYNQ CPU with 512 MB of DDR3 RAM.	667 MHz Dual Core ARM	0.148055	8.38x Slower than FPGA
PYNQ FPGA Xilinx xc7z020clg400-1	125 MHz	0.017659	2.43x Faster than Intel CPU 8.38x Faster than PYNQ ARM CPU

speedup of 8.38x compared to the feature extraction algorithm running solely on an arm processor.

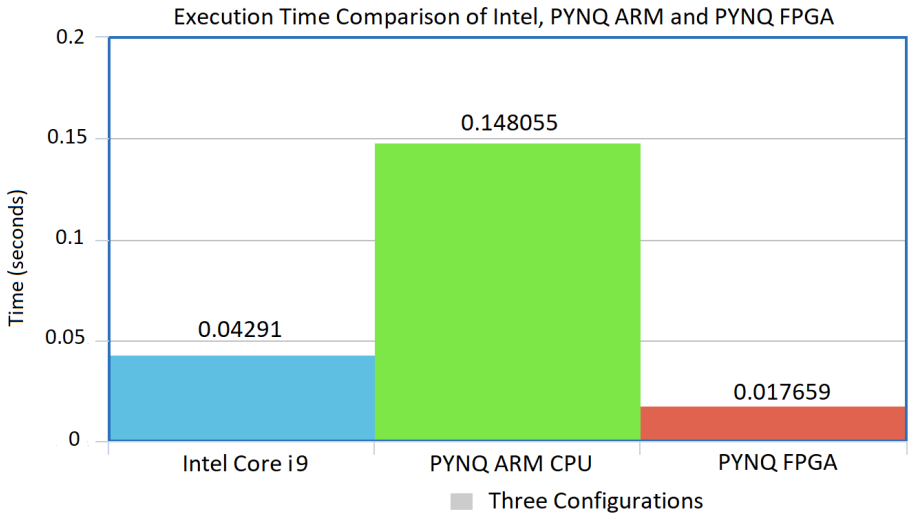

Figure 9. Execution Time Comparison using three platforms

5. Conclusion & Future Work

As system-on-chips become more heterogeneous and complex, a more software-oriented development environment is required. For software developers using Python to access the FPGA, Xilinx just launched PYNQ. A big step toward reaching a broader developer community, comparable to that of Raspberry Pi and Arduino, is being made possible by the combination of Python software and the parallel performance

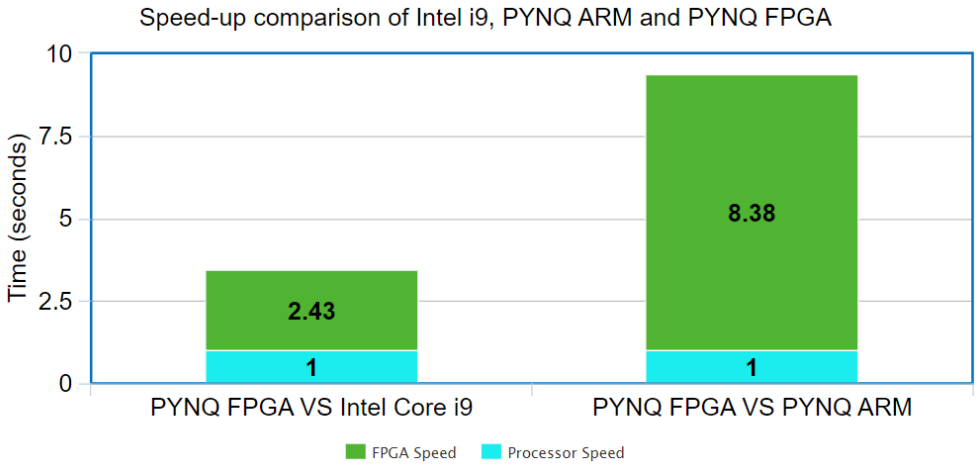

Figure 10. Speedup comparison using three platforms

capability of the FPGA. This research examined the performance of popular image processing methods like feature extraction for the visual SLAM in Python and specialized hardware accelerators to better understand the performance and capabilities of a Python and FPGA development environment. With a speedup of up to 8.38x, the results are very promising, with the ability to match and even exceed CPU performance in FPGA implementations. Furthermore, the results demonstrate that Python may still help software developers improve speed despite using extremely effective modules like OpenCV. This early research shows the operation of PYNQ and how Python communicates with the hardware accelerators and programmable fabric. The results are promising, so we are currently testing more algorithms in various visual SLAM-based image processing and machine learning domains. We will design, evaluate, and implement future hardware-accelerated feature matching for Visual SLAM.

6. Acknowledgements

This research was supported by Capacity Enhancement Program for Scientific a nd C ultural Exhibition Services through the National Research Foundation of Korea (NRF) funded by Ministry of Science and ICT (2018X1A3A106860331). We thank Professor Dongsoo Han from KAIST, who provided insight and expertise that greatly assisted the research, particularly the suggestions that greatly improved the manuscript. We would also like to thank the editors of the IPIN 2022 conference, Hong Yuan, Dongyan Wei, Wen Li and Antoni Pérez-Navarro for sharing their pearls of wisdom with us during this research.

References

[1] Bailey T *et al.* 2006 *IEEE Robotics & Automation Magazine* **13** 108 – 117 URL 10.1109/MRA.2006.1678144

[2] Lin S C 2018 *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*

[3] Yang S *et al.* 2017 *Robotics and Autonomous Systems* **93** 116–134 URL https://doi.org/10.1016/j.robot.2017.03.018

[4] Lategahn, Henning A, Geiger B and Kitt 2011 Visual SLAM for autonomous ground vehicles *2011 IEEE International Conference on Robotics and Automation, Shanghai, China* ed and others (IEEE) URL 10.1109/ICRA.2011.5979711

[5] Jeong W, Yeon K M and Lee 2006 Visual SLAM with line and corner features *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, 09-15 October 2006, Beijing, China* ed and others URL 10.1109/IROS.2006.281708

[6] Ni Q 2019 *Proceedings of the 2019 4th International Conference on Multimedia Systems and Signal Processing*

[7] Dumble S J and Gibbens P W 2015 *Journal of Intelligent & Robotic Systems* **78** 185–204

[8] Yun S 2018 *International Journal of Control, Automation and Systems* **16** 912–920

[9] Lowe G 2004 *International journal of computer vision* **60** 91–110

[10] Bay H, Ess A, Tuytelaars T and Van Gool L 2008 *Speeded-up robust features (surf)," Computer vision and image understanding* **110** 346–359

[11] Taketomi T, Uchiyama H and Ikeda S 2017 *Visual SLAM algorithms: A survey from 2010 to 2016* **9** 1–11

[12] Tertei D, Törtei J, Piat M and Devy 2016 *Computers & Electrical Engineering* **55** 123–137

[13] Sun R, Liu P, Wang J and Zhou Z 2017 *2017 IEEE International Symposium on Circuits and Systems (ISCAS* 1–4

[14] Park I and Kyu 2010 *IEEE Transactions* **22** 91–104

[15] Hajirassouliha A 2018 *Signal Processing: Image Communication* **68** 101–119

[16] Schmidt A G, Weisz G and French M 2017 *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*

[17] Lee D, Kim H and Myung H 2012 *2012 9th international conference on ubiquitous robots and ambient intelligence (urai)*

[18] Giubilato R *et al.* 2019 *Measurement* **140** 161–170 URL https://doi.org/10.1016/j.measurement.2019.03.038

[19] Peng T *et al.* 2020 *Electronic Imaging* **2020** 325–326 URL 10.2352/ISSN.2470-1173.2020.6.IRIACV-074

[20] Ma T 2021 *Wireless Communications and Mobile Computing* 2021–2021

[21] Tertei T, Piat J and Devy M 2014 *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)* 1–6

[22] Vourvoulakis J, Kalomiros J and Lygouras J 2016 *Microprocessors and Microsystems* **40** 53–73

[23] Fang W, Zhang Y, Yu B and Liu S 2017 *2017 International Conference on Field Programmable Technology (ICFPT* 275–278

[24] Ni Q 2019 *Proceedings of the 2019 4th International Conference on Multimedia Systems and Signal Processing*

[25] Mamri A *et al.* 2021 ORB-SLAM accelerated on heterogeneous parallel architectures *E3S Web of Conferences 229, 01055 (2021)* ed and others URL https://doi.org/10.1051/e3sconf/202122901055

[26] Imsaengsuk T, Pumrin S *et al.* 2021 Feature Detection and Description based on ORB Algorithm for FPGA-based Image Processing *2021 9th International Electrical Engineering Congress (iEECON)* ed and others pp 420–423 URL 10.1109/iEECON51072.2021.9440232

[27] PYNQ Z1 Development Boards URL http://www.pynq.io/board.html

[28] PYNQ Z1 Board URL https://www.student-circuit.com/news/learn-how-to-program-socs-with-pynq/

[29] PYNQ_Overlays URL https://pynq.readthedocs.io/en/v2.0/pynq_overlays.html

[30] Misra S, Wu Y *et al.* 2020 *Machine learning assisted segmentation of scanning electron microscopy images of organic-rich shales with feature extraction and feature ranking* URL 10.1016/b978-0-12-817736-5.00010-7

[31] Kallasi F, Rizzini D L and Caselli S 2016 *IEEE Robotics and Automation Letters* **1** 176–183

[32] Yang Y, Wang X, Wu J, Chen H and Han Z 2015 *The 27th Chinese Control and Decision Conference* 4996–4999

[33] Fan G, Xie Z C, Huang W, Cao L and Wang *IEEE Transactions on Circuits and Systems II: Express Briefs*

[34] Pham T H, Tran P and Lam S K 2019 *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **27** 747–756

[35] Pynq Board IO URL http://www.pynq.io/board.html

[36] Taranco R, Arnau J M and González A 2021 *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* 11–21