EE4002D Group Dissertation

Project Title:

Hardware Acceleration of Autonomous or Distributed Systems for Robotics

Team Member(s):

Chai Wei Lynthia (A0220888W)

Zubin Jain (A227713H)

Supervisor(s):

Associate Professor VADAKKEPAT, Prahlad

Dr Rajesh Chandrasekhara Panicker

College of Design and Engineering

In partial fulfilment of the Requirements for the Degree of Bachelor of Engineering

National University of Singapore

I. Acknowledgements

We extend our deepest appreciation to our esteemed Supervisors, Associate Professor VADAKKEPAT, Prahlad and Dr. Rajesh Chandrasekhara Panicker, whose guidance and expertise have been pivotal in shaping the trajectory of our group dissertation. Their mentorship and expertise have been instrumental in shaping the trajectory of our research endeavors.

Professor Prahlad's wealth of knowledge in robotics and insightful industry perspectives enriched our understanding and propelled our project to new heights.

Dr. Rajesh's adeptness at simplifying complex concepts ensured that we grasped intricate topics with ease, fostering an environment of intellectual curiosity and growth.

We are immensely grateful to Dr. Rethnaraj Rambabu for generously sharing his expertise in LIDAR and SLAM algorithms, which provided a solid foundation for our research endeavors. His insider insights into the robotics industry proved invaluable, offering us invaluable perspectives on current trends and challenges.

Additionally, we express our heartfelt gratitude to Esspressif Systems for their generous sponsorship of the ESP32-S3-LCD-EV board and ESP32-S3-DevKitC-1. Their support has been instrumental in realizing our project goals, and we eagerly anticipate showcasing our achievements in the Esspressif Capstone Project Competition.

Sincerely,

Chai Wei Lynthia and Zubin Jain

Contents

II. Figures

III. Tables

IV. List of Acronyms

A

API
Application Programming Interface 21

C

CPU
Central Processing Unit 33, 54
CSV
Comma-separated values 35, 36, 43

D

DMA
Direct Memory Access....................... 47, 48, 49, 50, 57

F

FPGA
Field Programmable Gate Arrays............................... 11
FSM
Finite State Machine 19

G

GUI
Graphical User Interface 35

H

HDL
Hardware Description Language................................ 19
HLS
High-Level Synthesis... 20, 45
HTTP
Hypertext Transfer Protocol....................................... 22

I

IMU
Inertial Measurement Unit 25
IoT
Internet of Things.. 18, 21, 60

L

LIDAR
Light Detection and Ranging 11, 12

O

OS
Operating System... 38

P

PF-SLAM
Particle Filter SLAM 24

R

RAM
Random Access Memory 18, 60
RTOS 38

S

SLAM
Simultaneous Localisation & Mapping................ 11, 12

U

ULP
Ultra Low Power.. 18, 19

V

VSLAM
Visual SLAM 23

1. Introduction

1.1 Problem Statement

Conventional LIDAR SLAM algorithms are known to have heavy computations on its processor. To accelerate the algorithms, different processors will require customisations and tweaking to reduce latency. This is because processors come with their own set of constraints which may require different workarounds.

Cost is one large factor in deciding the type of processor used to compute LIDAR SLAM algorithms and when cost is a factor, the constraints and specifications of processors will likely vary. Specifications affected by cost may include but are not limited to power, clock speed, threads, bandwidth, memory, etc.

If cost is not a constraint, we can afford to have a more powerful processor (E.g. larger memory, faster clock speed, acceleration on FPGA (Field Programmable Gate Arrays) fabric) where heavy computations can be done more efficiently. On one hand, the advantage is that heavy computations can be easily done. The disadvantage on the other hand is the high cost.

If cost is a constraint, a more budget-friendly processor is likely the choice while sacrificing certain aspects such as memory and clock speed where heavy computations have more limitations. The advantage is affordability, while the disadvantage is that heavy computations will be a little tricky from the engineer’s perspective. It is still possible to run calculations at a similar speed in comparison to a more sophisticated processor. Running certain heavy calculations on edge is an option if the processor is at its limits.

We will be designing various systems and accelerating them while considering different trade-offs associated with their processors. We will be evaluating them based on different configurations and system designs.

1.2 Value Proposition

1.2.1 Stakeholders and Their Needs

Companies interested in comparing various implementation strategies of hardware acceleration on LIDAR SLAM can evaluate their decisions based on metrics such as cost, speed, bandwidth etc.

As different companies have different constraints and use cases of LIDAR SLAM, this is useful when deciding which implementation is most suitable for LIDAR SLAM based on their own use cases.

1.2.2 Approach to Problem

After designing various systems while dealing with different trade-offs associated with its processors, we intend to profile the system to check for its bottleneck to accelerate the overall system for higher throughput.

2. Aim

2.1 Final Aim

The aim of this project is to accelerate LIDAR SLAM algorithms on differing processors of contrasting price points, compare their differences and weigh their advantages and disadvantages.

3. Functional Block Diagram

Figure 1 shows the overall block diagram of the system to be designed for differing processors.

- Subsystem 1 (member 1, Chai Wei Lynthia): This subsystem encapsulates the main robotics algorithm which processes LIDAR scans. Refer to Figure 2.
- Subsystem 2 (member 2, Zubin Jain): This subsystem encapsulates the hardware implementation and interface for feeding data into system 1 for differing processors as well as the external server.

3.1 Overall Functional Block Diagram

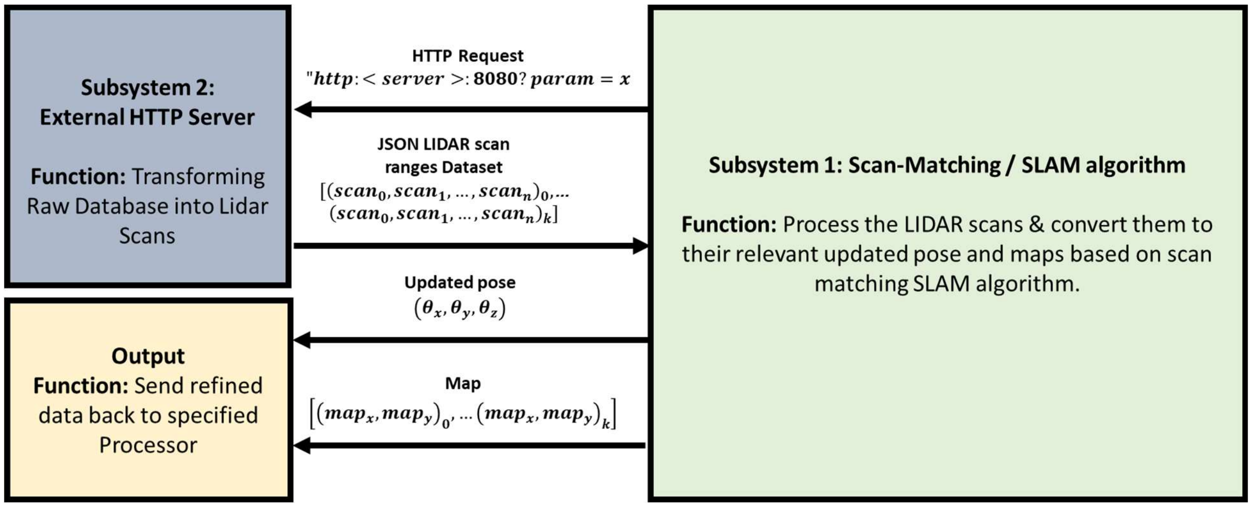

Subsystem 2: External HTTP Server
Function: Transforming Raw Database into Lidar Scans
Output
Function: Send refined data back to specified Processor
HTTP Request
"$http:< server >: 8080? param = x$
JSON LIDAR scan ranges Dataset
$[(scan_0, scan_1, \dots, scan_n)_0, \dots (scan_0, scan_1, \dots, scan_n)_k]$
Updated pose
$(\theta_x, \theta_y, \theta_z)$
Map
$[(map_x, map_y)_0, \dots (map_x, map_y)_k]$
Subsystem 1: Scan-Matching / SLAM algorithm
Function: Process the LIDAR scans & convert them to their relevant updated pose and maps based on scan matching SLAM algorithm.

Figure 1: Overall Functional Block Diagram

3.2 Subsystem 1 – Overview

Figure 2 showcases the high-level implementation of Subsystem 1. Scan matching SLAM can be broken up into a multitude of functions.

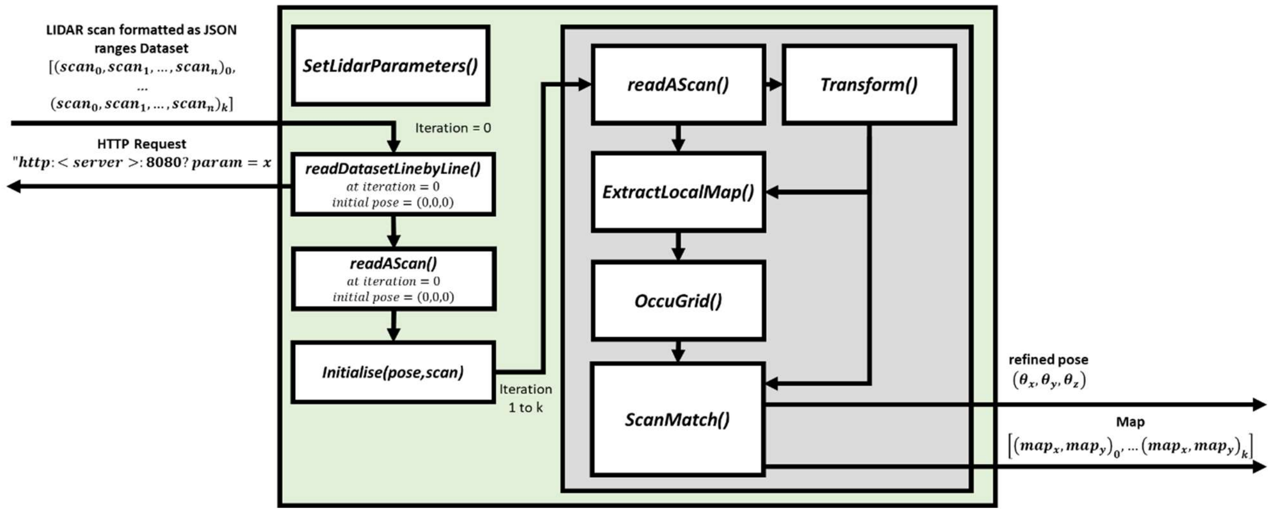

LIDAR scan formatted as JSON ranges Dataset
$[(scan_0, scan_1, \ldots, scan_n)_0, \ldots (scan_0, scan_1, \ldots, scan_n)_k]$
HTTP Request
$"http:< server >:8080?param = x$
SetLidarParameters()
Iteration = 0
readDatasetLinebyLine()
$at\ iteration = 0$
$initial\ pose = (0,0,0)$
readAScan()
$at\ iteration = 0$
$initial\ pose = (0,0,0)$
Initialise(pose,scan)
Iteration 1 to k
readAScan()
Transform()
ExtractLocalMap()
OccuGrid()
ScanMatch()
refined pose
$(\theta_x, \theta_y, \theta_z)$
Map
$[(map_x, map_y)_0, \ldots (map_x, map_y)_k]$

Figure 2: Subsystem 1 (Overview)

3.3 Subsystem 2 – Overview

Figure 3 showcases the high-level implementation of Subsystem 2.

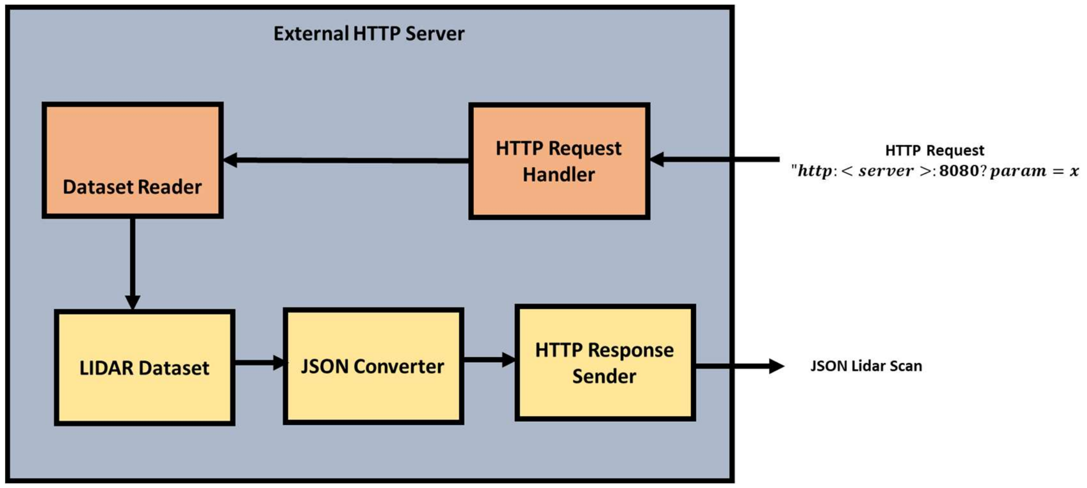


Figure 3: Subsystem 2 (Overview)

4. Timeline

4.1 Rough Overall Timeline of Semester 1 and 2 (Gantt Chart)

2023 (Semester 1)							2024 (Semester 2)			
Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr
Research phase										
	Software Simulation/Realisation in C									
				Code Optimisation						
						Hardware implementation of Kria/ESP32				
										Submission month

Table 1: Capstone Overall Timeline 2023/24

4.2 Detailed Timeline of Semester 1

Semester 1 Timeline		
Summer Break	**Ideation of Overall Architecture of Capstone** • Discussed different variations of architecture. • Feasibility checks of overall Architecture with Supervisors	
Week 1	**Group**	• Decided on specific design architecture • Literature Review Progress on existing hardware acceleration for SLAM • Reviewed Literature on both Graph and Particle SLAM algorithms
	Subsystem 1	• **Current Progress / functions:** ➔ LIDAR dataset: Dataset of 5522 rows x 1079 columns ✓ **rows:** no. of scan ranges ✓ **columns:** no. of point clouds per scan range ➔ Scan function: LIDAR to global scan data in C ✓ (scan_x, scan_y) ➔ Initialise function: map points, pose & scan
	Subsystem 2	• Explored various robotics algorithms, attempting implementation of Kalman Filter Slam
Week 2/ Week 3	**Group**	• Work on Individual Algorithms
	Subsystem 1	**Occupational grid map function for LIDAR scan matching** • Create occupancy grid map • Create metric map based on occupancy grid map (based on Euclidean distance transform)
	Subsystem 2	• Developed connection between ESP32 and LIDAR Server
Week 4	**Group**	• Optimise Necessary Functions
	Subsystem 1	• Optimise Occupational grid map function
	Subsystem 2	• Continued with exploration of robotics algorithms as well research on various possible embedded systems
Week 5	**Group**	• Refine / update outputs of functions based on previous weeks
	Subsystem 1	• Experiment with different specifications such as pixel sizes to refine outputs

		• Start CA1 report
	Subsystem 2	• Continued with exploration of SLAM algorithms as well research on various possible embedded systems
Week 6	**Group**	• first draft of CA1
Reading Week / Week 7	**Group**	• Exam period
Week 8	**Group**	• Work on Individual Algorithms • Returned to exploring possible mapping implementation on the ESP32
	Subsystem 1	• Course scan matching function for LIDAR data ➔ Experiment with different resolution sizes
Week 9	**Group**	• Work on Individual Algorithms
	Subsystem 1	• Fine scan matching function for LIDAR data
	Subsystem 2	• Work on CA1 report
Week 10	**Group**	• Work on Individual Algorithms
	Subsystem 1	• Work on CA1 report
	Subsystem 2	• Concluded that actual mapping implementation on the ESP32 was impossible; started exploring edge computing
Week 11	**Group**	• Buffer Week
Week 12	**Group**	• Finalise / Submit CA1 report

Table 2: Semester 1 Timeline

4.3 Detailed Timeline of Semester 2

Semester 2 Timeline		
Winter Break	**Running Code on ESP32 with Edge Computation** • Assimilate code from simulation to run on the processor. • Edge computation: E.g. running on laptop for proof of concept	
Week 1 – Week 5	**Subsystem 1**	**Running Code on ARM Processor** • Profile • **Priority:** get the LIDAR SLAM code running on ARM processor • Profile code for FPGA acceleration (check if FPGA copressor is needed or need)
	Subsystem 2	• Started work on ESP32 network interface; explored in-built demo apps for suitable protocols for edge processing. • **Created Lidar simulator on desktop** to help with ESP32 interfacing. • Integrated the ESP32 with the python LIDAR simulator
Week 6- Week 12	**Group**	• **Documentation of Code and Report Update**
	Subsystem 1	• **Validate and test FPGA Co-processor**

Table 3: Semester 2 Timeline

5. Overall System – Processors Experimented with

Board 1: Kria AI starter kit [1]

- Based on Zynq UltraScale+ MPSoC chip
- Includes ARM Cortex-A53 Quad core, 2 ARM Cortex-R5 cores, and FPGA fabric.
- If cost is not a huge constraint, we can have a larger processor such as Board 1 which comes integrated with FPGA fabric. Compared to GPUs, FPGAs have lower power consumption making it suitable for embedded systems. It is also extremely customisable where we can incorporate our own accelerator.

Board 2: ESP32-S3-DevKitC-1 [2]

- Based on ESP32-S3
- Includes Xtensa Dual-core 32-bit LX7 Microprocessor
- If cost is a constraint, we can have more budget friendly processors such as Board 2. Unlike Board 1, Board 2 is more affordable with a plethora of smaller scale features.

Note: Subsystem 1 and Subsystem 2 is implemented into Board 1 and Board 2

5.1 Comparison Table of Boards

Figures of Merits	**Kria AI Starter Kit (Board 1)**	**ESP32-S3-DevKitC-1 (Board 2)**
Cost	US $249 MSRP [1] ≈SGD $340.81	SGD $22.19 [3]
	• Board 1 cost approximately 15 times more than that of Board 2	
Acceleration Method	FPGA Fabric (High-Level Synthesis)	Accelerated with Edge (A nearby computer/server)
Target Application	Designed for the purpose to accelerate	Primarily intended for IoT (Internet of Things)
In-Built Coprocessor	Customisable Coprocessor (based on FPGA fabric)	Ultra Low Processor (ULP)
Memory	4GB of RAM	52kBytes of RAM

Table 4: Comparison Table of Boards (Different Processors)

5.2 Comparison of Different Processor Boards

Figure 4 shows the Functional Block Diagram of ESP32-S3-DevKitC-1. Inside the Esspressif board contains an in-built co-processor named the ULP (Ultra Low Power) Processor. According to [4], the ULP is a simple finite state machine (FSM) meant to be used while the main processors are in Deep-sleep mode. It is simply not meant for hardware acceleration.

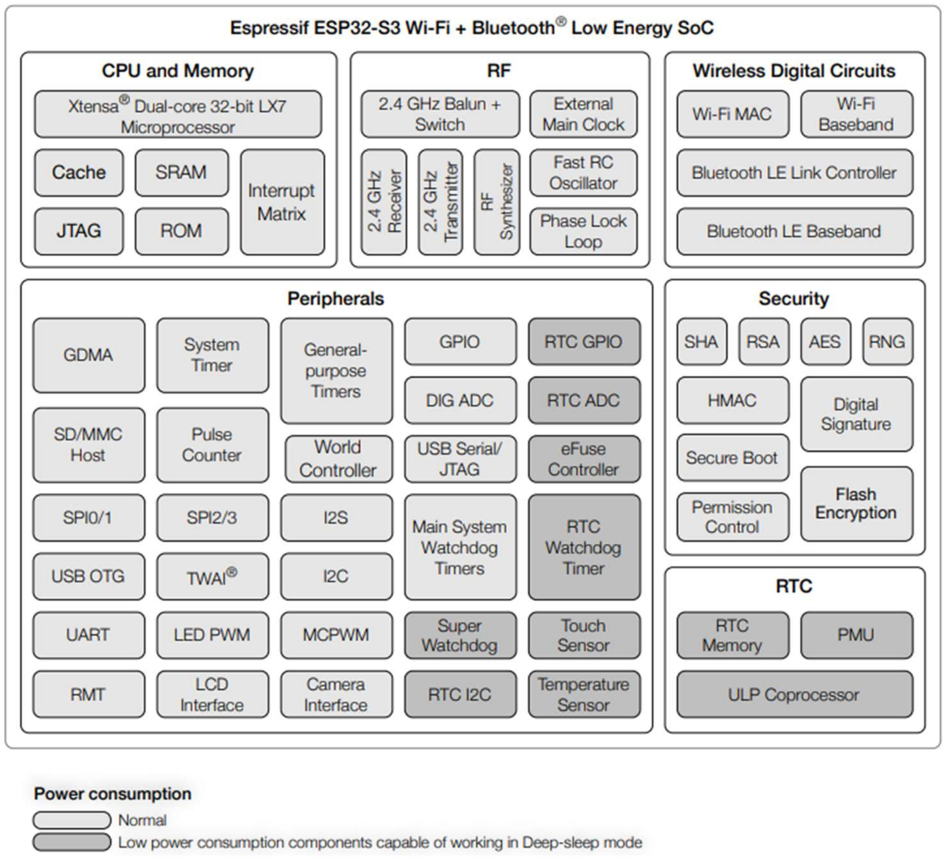


Figure 4: Functional Block Diagram of ESP32-S3 Series (Courtesy of [5])

Figure 5 shows the Functional Block Diagram of Kria AI Starter Kit. Inside the KV260 contains FPGA fabric which can be accelerated using High-Level Synthesis of Software code or writing it in Hardware Description Languages (HDL).

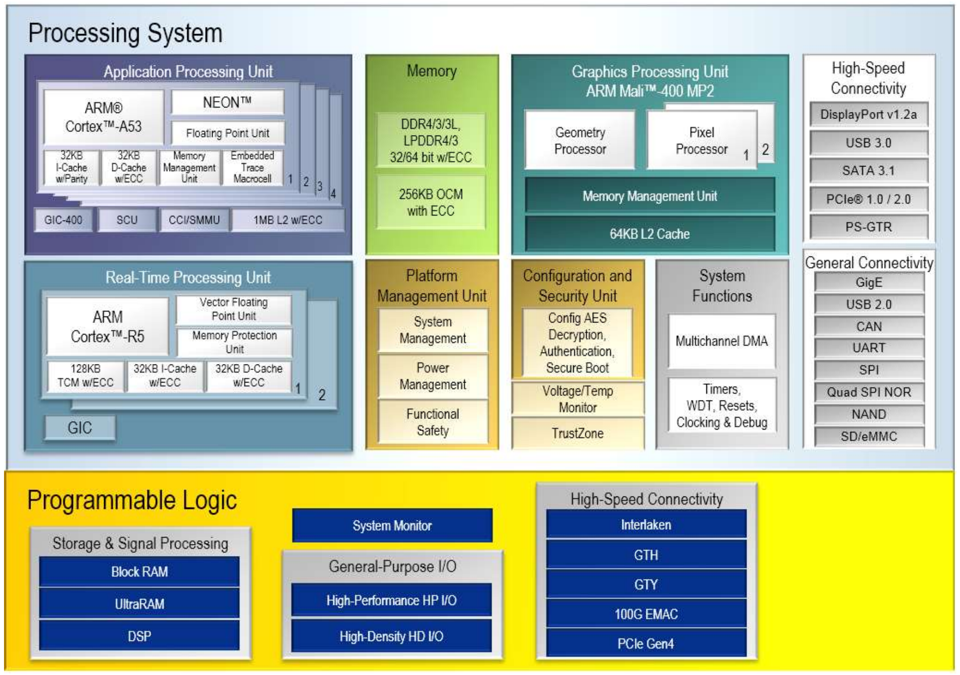


Figure 5: Functional Block Diagram of Kria AI Starter Kit (Courtesy of [6])

6. Overall System – Chosen Acceleration Methods on Processors

6.1 Acceleration on Kria AI Starter Kit

Since FPGA fabric is included in the KV260 board, it would be beneficial to make use of the FPGA fabric for potential hardware acceleration. Although we have a choice between running on the hardware level such as writing code in VHDL or Verilog, running on High-Level Synthesis (HLS) such as writing code in C/C++ would be beneficial as writing in HDL is too time consuming.

Although offloading certain heavy computations on HLS is not as optimized as writing on HDL, the time taken is shortened by a large margin. We have to take into account that many robotics engineers have little to no contact with FPGAs in general and not to mention the steep learning curve of learning digital logic design on FPGAs.

Furthermore, writing in HDL can potentially optimize the code further, but the results are diminishing if we consider a much longer time taken.

Hence, considering the constraint of time taken, offloading some heavy computations on HLS on FPGA is the most suitable solution on the KV260 board.

6.2 Acceleration on ESP32-S3-DevKitC-1

The Xtensa microprocessor within ESP32 is considered lightweight and meant for IoT applications. Running heavy computations like LIDAR SLAM can be a challenge especially if memory is not enough.

Additionally, the in-built co-processor on ESP32-S3-DevKitC-1 is also not meant for heavy computations like LIDAR SLAM, the only way we can accelerate the computations on ESP32 is to offload the computations to edge. ESP32 will essentially function like an Application Programming interface (API).

Rather than accelerating embedded like the KV260 board, we could possibly run the heavy computations on Edge to accelerate the process. ESP32 can then do what it is meant to do, which is IoT applications without compromising computation accuracy.

6.3 Programming Language Used

C language is easily integrable for ESP32-S3-DevKitC-1 and Kria AI starter kit. Moreover, C is known for its excellent performance in applications where speed is essential.

Python was used for the LIDAR server and for the PYNQ hardware drivers. In the first case it was used due to the language's simplicity with regards to web-server and low-importance of performance in that application (since LIDAR sensors typically operate around 10Hz) while for the latter it was used due to being the only language supported by the framework.

Figure 8 shows a simulation of LIDAR SLAM of the Deutsches Museum on MATLAB.

6.4 Documentation of Code

Refer to https://github.com/circuitpotato/Hardware-Acceleration-of-LIDAR-SLAM.git for the actual implementation of Subsystem 1 and Subsystem 2.

7. Subsystem 1 – Legend

Refer to Figure 2 for more details regarding Subsystem 1 overview.

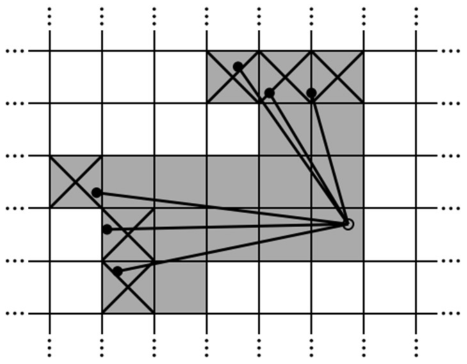
Terms	Definition
HTTP Request	Purpose: Subsystem 1 will send a HTTP request to a webserver encapsulated within Subsystem 2 for every LIDAR scan iteration.
Lidar scan ranges dataset: $\begin{bmatrix}(scan_0, scan_1 \ldots)_0, \ldots \\ ,(scan_0, scan_2)_{3480}\end{bmatrix}$	Purpose: when the HTTP request is successful, it will send over 1 iteration for every HTTP request (equivalent to **1079** LIDAR point clouds). Input matrix dataset: $\mathbf{3480} \times \mathbf{1079}$. (~15MBytes) Row (3480): no. of scan ranges → 3480 iterations Column (1079): no. of scans per iteration
Refined pose: $(\theta_x, \theta_y, \theta_z)$	Pose is refined by scan matcher from subsystem 1
Transformed scan: (r_k, θ_k)	Transform the **1079** scans of each iteration to polar form (world points in world frame)
Pixel hits / submap: $(hit_0, .., hit_k)$	 *Figure 6: Scan and Pixels associated with hits (courtesy of [7])*
Updated pose: $(\theta_x, \theta_y, \theta_z)$	Updated pose is the pose updated from the SLAM algorithm from Subsystem 1.
Map: $\begin{bmatrix}(\boldsymbol{map_x}, \boldsymbol{map_y})_{\mathbf{0}}, \ldots \\ ,(\boldsymbol{map_x}, \boldsymbol{map_y})\end{bmatrix}$	This is the updated map as produced by Subsystem 1 in world frame.

Table 5: Subsystem 1 Legend

8. Subsystem 1 – Preliminary Processes

This portion describes the ***rationale*** and ***reason(s)*** behind all decisions within the scope of Subsystem 1.

8.1 Choice of LIDAR SLAM vs VSLAM

LIDAR SLAM uses lasers while VSLAM uses visual sensors like cameras to capture raw data. According to [8], Although vision-based approaches (E.g., VSLAM) is getting more popular, Vison sensors (E.g., cameras) are sensitive to unpredictable changes of the environment such as change in lighting whereas Laser Range Finder (LRF) scans are still widely desired due to robustness towards noise. In addition, most VSLAM approaches are not sufficiently accurate for autonomous robot navigation.

This means that LIDAR SLAM has the advantage of accuracy compared to VSLAM. Hence, we decided to proceed with LIDAR SLAM approach. Although VSLAM in general is technically cheaper as compared to LIDAR SLAM to implement (because cameras are relatively inexpensive compared to LIDARs), its accuracy can be affected by various environmental conditions whereas LIDARs are less affected by the appearance of the environment.

8.2 LIDAR Dataset

Due to the expensive nature of LIDARs, we will be simulating datasets instead of using the actual component. The standard LIDAR sensor operates at 10Hz for data acquisition of scan ranges. This means it takes 10 scans per second.

8.2.1 Real World Dataset

We will be working on a LIDAR dataset of the Deutsches Museum. Since I am referencing data and algorithm calculations from [7], it is beneficial to use one of their example maps.

Note: Due to the vast size of Deutsches Museum, we will only be simulating a portion of the museum for efficient prototyping (or simulation). The dataset used is based on [9].

8.3 Choice of Scan Matching

8.3.1 What is Scan Matching?

It is a technique used in SLAM to estimate a robot's position or transform between two robot positions where scans were taken [10]. Scan matching tries to incrementally align two scans or map to a scan without revising the map.

8.3.2 Why use Scan Matching?

According to [8], scan matching plays an essential role to efficiently solve SLAM and by matching sensor scans taken from different poses, scan matching can efficiently estimate rigid transformation of the robot between 2 poses. This means that scan matching can effectively localise the robot with respect to the input data scans or maps.

Scan matching is one of the fundamentals of SLAM algorithms as shown in Figure 7. SLAM algorithms are generally chosen between particle filter SLAM (PF-SLAM) and graph-based SLAM [11]. Both PF-SLAMs and Graph-based SLAMs can incorporate scan matching. Thus, it motivates the choice of scan matching.

Essentially, we can think of scan matching as a form or part of SLAM. We only need to code out the scan matching algorithm once and we can modify our SLAM algorithms to suit the situation.

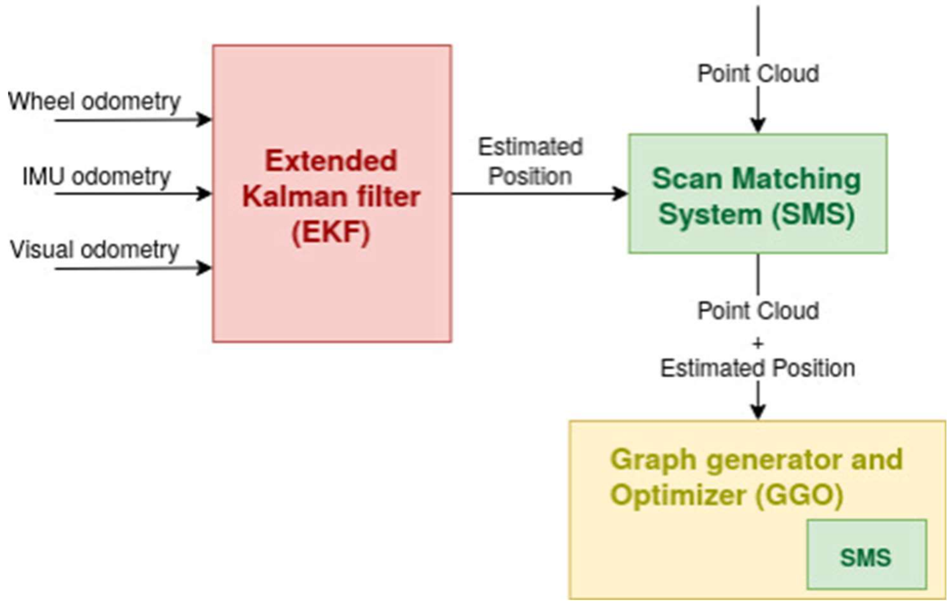

Wheel odometry
IMU odometry
Visual odometry
Extended Kalman filter (EKF)
Estimated Position
Point Cloud
Scan Matching System (SMS)
Point Cloud + Estimated Position
Graph generator and Optimizer (GGO)
SMS

Figure 7: Structure of typical SLAM system (Courtesy of [12])

In Figure 7, The Extended Kalman Filter black box is simply a way to calculate the IMU odometry (motion commands, distance, velocity motion model). For simplicity, we will assume a constant velocity motion model. Because

8.3.3 Types of Scan Matching

In general, there is a multitude of ways to implement scan matching where each type of implementation has its own perks and trade-offs. Some examples include Iterative Closest Point (ICP), scan-to-scan matching, scan-to-map matching, map-to-map matching, feature based and much more.

8.3.4 Choosing scan matching type – scan-to-map

Feature based matching might seem plausible at first glance, but it requires distinct features as its name suggests. Thus, if the environment or map has does not have distinguishable or obvious features, it can heavily affect the accuracy of the map.

ICP is a popular choice amongst SLAM algorithms and is often incorporated into scan-to-scan matching and scan-to-map matching. Scan-to-scan matching is also another plausible choice, but scan-to-scan quickly accumulates error and scan-to-map matching helps limit the accumulation of error [7]. This means that scan-to-map is less susceptible to drift or error in the map with more incoming LIDAR data.

8.3.5 Subsystem 1 – Simulation on MATLAB

Before implementing in C, simulating on MATLAB is an efficient way to check for the correctness of my pseudocode. Furthermore, it is easier to visualise the output map of LIDAR SLAM. Once the algorithm correctness is ensured, we can start implementing it in C language.

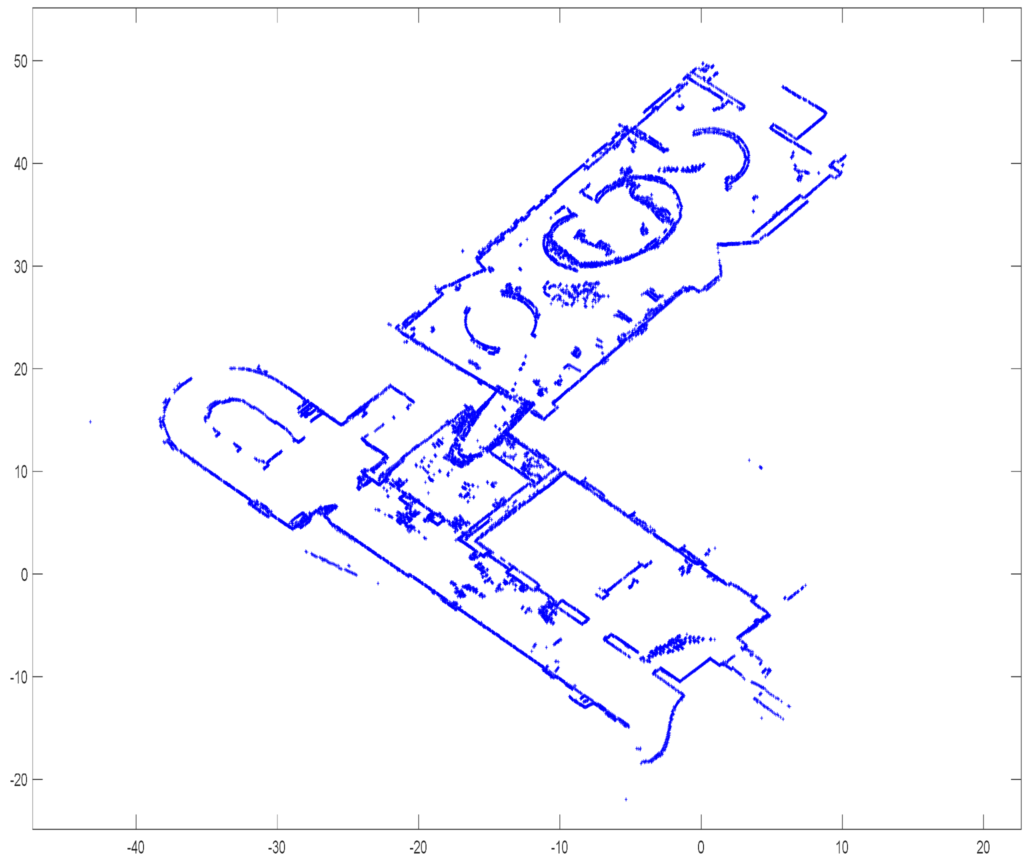


Figure 8: Scan Matching Simulation on MATLAB (modified simulation code, courtesy of [13])

9. Subsystem 1 – Algorithm Pseudocode

To make the overall system neater, we will be evaluating our algorithms in functions for both processors. As both Kria KV260 and ESP32-S3 will be incorporating similar algorithms with the same programming language, we can evaluate the algorithms of the C code together, followed by customising the algorithms to suit different boards constraints individually.

Note: The Algorithm is based on the simulation [13] in MATLAB and the research paper [7]. However, there are a lot of differences implemented in C language since both languages are inherently different by nature. Naming conventions are kept similar for easier reference and said references are only used as a benchmark, but they are still coded very differently in C.

Only the main logic is explained below. There might be slight differences in raw code.

9.1 Set LIDAR Parameters

Function 1: SetLidarParameters

```
1. lidar.angle_min = …
2. lidar.angle_max = ..
3. /*
4.    other parameters
5. */
6. lidar.angles = …
```

Purpose: *Because we do not have an actual LIDAR sensor, we must specify our own LIDAR values. All values are referred to from* [9]. *All values are measured in radians.*

- **Angle min / max:** the min/max rotates $360°$
- **Angle increment:** $angle_{min} + (npoints - 1)(angle_{increment}) = angle_{max}$

  E.g. $angle = [angle_{min}, angle_{min+increment}, .., angle_{max}]$

9.2 Reading A Scan

Function 2: ReadAScan

```
1. for scan = 0 to scan_size - 1 do
2.    if scan > min_lidar_range then
3.       if scan > max_lidar_range then
4.          Continue
5.     else // save clean scan x and scan y
6.       end if
7.    end if
8. end for
9. return scan
```

Purpose: *This function cleans up each iteration's scan range based on the lidar specified LIDAR parameters. Removing outliers can prevent accumulation of error.*

- This function takes in the raw LIDAR data **scan** in the form of **scan[idx][i]** where **idx** is the iteration of scan ranges while **i** is the scan of each scan ranges. If **scan[idx][i]** is an outlier, it will automatically be removed.
- Finally, it will convert the clean scans from polar to cartesian.

9.3 Transform

Function 3: Transform

```
1. pose = $\xi_x, \xi_y, \xi_\theta$
2. $T_\xi p = \begin{pmatrix} \cos\xi_\theta & -\sin\xi_\theta \\ \sin\xi_\theta & \cos\xi_\theta \end{pmatrix} p + \begin{pmatrix} \xi_x \\ \xi_y \end{pmatrix}$
3. **return** $T_\xi p$
```

Purpose: *Transform scan range & pose of each iteration into world points in the global frame*

- It is a sub function of **ExtractLocalMap()** and **Initialise()**
  ➔ $\boldsymbol{\xi_x, \xi_y, \xi_\theta}$ is represented as $\boldsymbol{\cos\theta, \sin\theta, \theta}$ in the code snippet.
  ➔ $\boldsymbol{p}$ is the x and y coordinates of the scan represented by $\boldsymbol{scan.x[i]}$ and $\boldsymbol{scan.y[i]}$ in the code snippet.

9.4 Extract Local Map

Function 4: ExtractLocalMap

```
1.  pose = $\xi_x, \xi_y, \xi_\theta$
2.  $T_\xi p$ =Transform(scan)
3.  minX = min($T_\xi p$) - borderSize
4.  minY = min($T_\xi p$) - borderSize
5.  maxX = max($T_\xi p$) + borderSize
6.  maxY = max($T_\xi p$) + borderSize
7.  for i = 0 to i = map_size – 1 do
8.    if $minX < map_x < maxX$ then
9.      if $minY < map_y < maxY$ then
10.       $localMap_x = map_x$
11.       $localMap_y = map_y$
12.     end if
13.   end if
14. end for
15. return localMap
```

Purpose: *extract a local map based on the current scan iteration*

- **Transform()** is used here as a sub function to convert the current scan into the world frame
- **minX**, **minY**, **maxX**, **maxY** are used to set the top-left & bottom-right corner of the local Map. Only if the most updated map points are encapsulated within said constraints will the transformed scan be updated as part of the local map.

9.5 Occupational Grid

Function 5: OccupationalGrid

```
1.  $minX = \min(localMap_x)$
2.  $minY = min(localMap_y)$
3.  $maxX = \max(localMap_x)$
4.  $maxY = \max(localMap_y)$
5.  $gridSize_x = \frac{maxX - minX}{pixelSize}$,  $gridSize_y = \frac{maxY - minY}{pixelSize}$
6.  $hits_x = \frac{localMap_x - minX}{pixelSize}$, $hits_y = \frac{localMap_y - minY}{pixelSize}$
7.  for i = 0 to i = localMapSize do
8.      $grid_x = hits_y * gridSize_x + hits_x$,
9.      $grid_y = hits_x * gridSize_y + hits_y$,
10. end for
11. metricGrid = euclideanDistanceTransform(grid)
12. return $grid(x, y)$ and $metricGrid(x, y)$
```

Purpose: *Create an occupancy grid map from the local map where it is made up of 0's and 1's array.*

- The minimum and maximum x and y values are calculated similarly to **ExtractLocalMap()**.
- The grid size is calculated based on outputs of **ExtractLocalMap()**.
- **minXY** and **maxXY** are used to set the top-left & bottom-right corner of the grid Map. Only if the most updated map points are encapsulated within said constraints will the transformed scan be updated as part of the local map.
- **metricGrid** is solved using Euclidean distance as mentioned below.

9.6 Euclidean Distance Transform

9.6.1 Naïve Euclidean Distance Transform Function

Function 4: Naïve Euclidean Distance Transform

```
1.  height, width
2.  for y = 0 to y = height do
3.    for x = 0 to x = width do
4.      if $grid[y][x] \neq 0$ then
5.        $metricGrid[y][x] = 0$
6.      else
7.        for j = 0 to j = height do
8.          for i = 0 to I = width do
9.            if $grid[j][i] \neq 0$ then
10.             $distance = \sqrt{(x-i)^2 + (y-j)^2}$
11.           end if
12.         end for
13.       end for
14.     end if
15.   end for
16. end for
```

Purpose: *distance transform of a binary image.*

- **Note:** This code is based on the Euclidean distance equation provided for **bwdist()** function of MATLAB [14].
- It is a sub function of **OccupationalGrid()** (courtesy of [14])

9.6.2 Optimised Euclidean Distance Transform Function

Function 4: Optimised Euclidean Distance Transform

```
height, width
maxDist = 10
for y = 0 to y = height do
  for x = 0 to x = width do
    if $grid[y][x] \neq 0$ then
      $metricGrid[y][x] = 0$
    else
      minDist = maxDist
      for j = 0 to j = height do
        for i = 0 to I = width do
          if $grid[j][i] \neq 0$ then
            $squaredDistance = (x - i)^2 + (y - j)^2$
            if $squaredDistance < minDist^2$ then
              $minDist = \sqrt{squaredDistance}$
            end if
          end if
        end for
      end for
    end if
  end for
end for
```

Purpose: *distance transform of a binary image.*

- **Purpose of Limiting Distance to 10 units**
  - ➔ This reduces the range of values to calculate considering that we have large datasets to work with. It is more computationally efficient.
  - ➔ It is common to limit the distance when working with Euclidean transform because we want to limit the range for easier interpretation.

- **Purpose of Minimising Square Root Calculations**
  - ➔ Square root calculation is known to slow down calculations due to its non-linear nature.
  - ➔ Calculating squared distances (essentially just multiplication) can reduce latency in the long run, especially when said function is used every iteration.
  - ➔ A simple way to reduce square root calculations is enforcing a simple conditional statement to limit the calculations to its minimum.

- **Euclidean Distance Function is based on bwdist() in MATLAB**

9.7 Scan Matching

Function 5: ScanMatch

```
1.  maxIter, maxDepth, iteration, depth
2.  $pixelScan = scan \times pixel$
3.  while iteration < maxIter do
4.    noChange is True
5.    for $[\theta - r, \theta, \theta + r]$ do
6.      for $[pose_x - t, 0, pose_x + t]$ do
7.        for $[pose_y - t, 0, pose_y + t]$ do
8.          /* calculate score from hits*/
9.          if score < bestScore then
10.           noChange is False
11.           $bestPose = [x, y, \theta]$
12.           bestScore = score
13.         end if
14.       end for
15.     end for
16.   end for
17.   if noChange is True then
18.     increase resolution
19.   end if
20. end while
```

Purpose: *Very simple scan matching code to align the LIDAR scans with the map to estimate a refined pose.*

- Google’s Cartographer provides a real-time solution for indoor mapping in the form of a sensor equipped backpack that generates 2D grid maps with a resolution r = 5 cm according to [9]. Hence, resolution r will start initially with r = 0.05.
- **If noChange is True:**
  - ➔ This means that there is no better match found. Hence, we need to increase our resolution. We can do this where resolution **r = r / 2**.
  - ➔ **t** is the is the resolution or degree rotation of $pose_x$ and $pose_y$. it will similarly need to increase resolution when no better match is found.

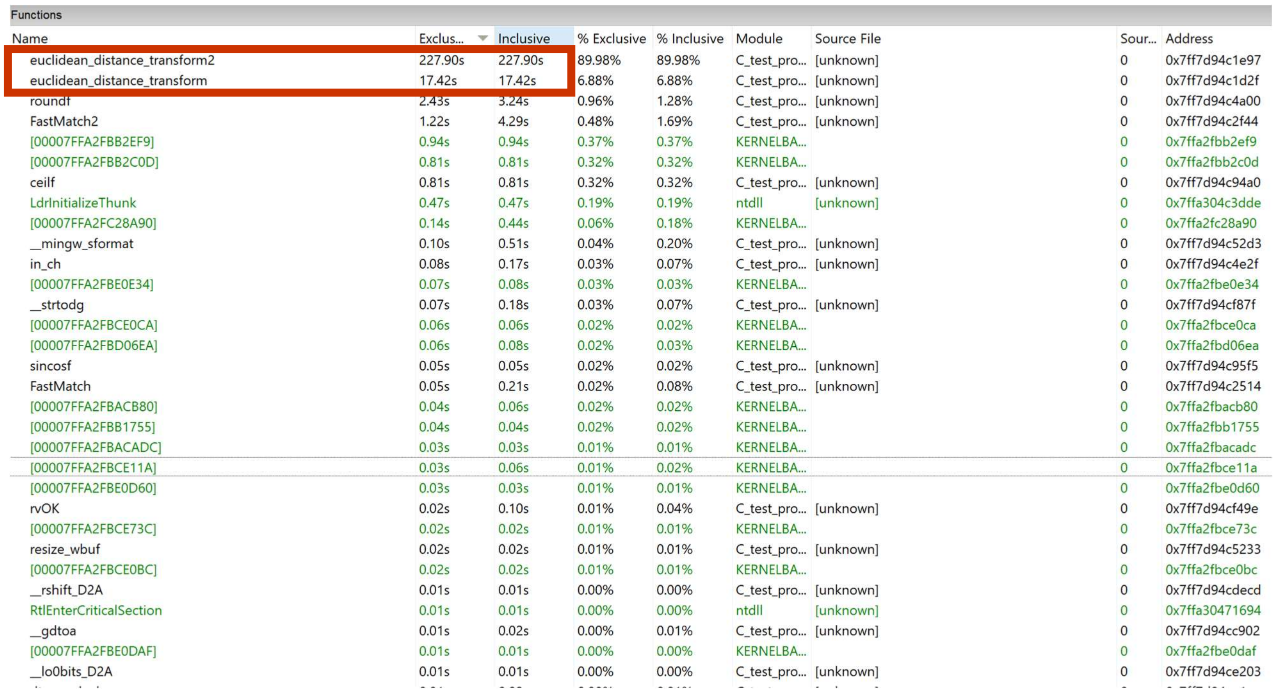
10. Overall System – Profiling Code

Once all the C code is fully written, the next step is to profile the code. For simplicity, we will be profiling code for 1000 incoming LIDAR Scans in float format (1000 rows by 1079 columns, approximately 4.3MBytes).

“Very Sleepy”, an open-source Central Processing Unit (CPU) profiler is used [15]. Profiling results are displayed in Figure 9 as shown below. It is evident that the bottleneck is the Euclidean Distance Transform Function as mentioned in point 9.6.

Despite minimizing the distance as mentioned and reducing square root calculations on the function, the highest timing required (227 seconds) to execute the function is still more than that of other functions. Hence, it makes sense to ***accelerate the Euclidean distance transform function or its partial function***.

$$Total\ Software\ Timing\ Profile = \frac{227.9}{89.98} \times 100 = \underline{\boldsymbol{253.28\ s}}$$



Functions

Name	Exclus...	Inclusive	% Exclusive	% Inclusive	Module	Source File	Sour...	Address
euclidean_distance_transform2	227.90s	227.90s	89.98%	89.98%	C_test_pro...	[unknown]	0	0x7ff7d94c1e97
euclidean_distance_transform	17.42s	17.42s	6.88%	6.88%	C_test_pro...	[unknown]	0	0x7ff7d94c1d2f
roundf	2.43s	3.24s	0.96%	1.28%	C_test_pro...	[unknown]	0	0x7ff7d94c4a00
FastMatch2	1.22s	4.29s	0.48%	1.69%	C_test_pro...	[unknown]	0	0x7ff7d94c2f44
[00007FFA2FBB2EF9]	0.94s	0.94s	0.37%	0.37%	KERNELBA...		0	0x7ffa2fbb2ef9
[00007FFA2FBB2C0D]	0.81s	0.81s	0.32%	0.32%	KERNELBA...		0	0x7ffa2fbb2c0d
ceilf	0.81s	0.81s	0.32%	0.32%	C_test_pro...	[unknown]	0	0x7ff7d94c94a0
LdrInitializeThunk	0.47s	0.47s	0.19%	0.19%	ntdll	[unknown]	0	0x7ffa304c3dde
[00007FFA2FC28A90]	0.14s	0.44s	0.06%	0.18%	KERNELBA...		0	0x7ffa2fc28a90
__mingw_sformat	0.10s	0.51s	0.04%	0.20%	C_test_pro...	[unknown]	0	0x7ff7d94c52d3
in_ch	0.08s	0.17s	0.03%	0.07%	C_test_pro...	[unknown]	0	0x7ff7d94c4e2f
[00007FFA2FBE0E34]	0.07s	0.08s	0.03%	0.03%	KERNELBA...		0	0x7ffa2fbe0e34
__strtodg	0.07s	0.18s	0.03%	0.07%	C_test_pro...	[unknown]	0	0x7ff7d94cf87f
[00007FFA2FBCE0CA]	0.06s	0.06s	0.02%	0.02%	KERNELBA...		0	0x7ffa2fbce0ca
[00007FFA2FBD06EA]	0.06s	0.08s	0.02%	0.03%	KERNELBA...		0	0x7ffa2fbd06ea
sincosf	0.05s	0.05s	0.02%	0.02%	C_test_pro...	[unknown]	0	0x7ff7d94c95f5
FastMatch	0.05s	0.21s	0.02%	0.08%	C_test_pro...	[unknown]	0	0x7ff7d94c2514
[00007FFA2FBACB80]	0.04s	0.06s	0.02%	0.02%	KERNELBA...		0	0x7ffa2fbacb80
[00007FFA2FBB1755]	0.04s	0.04s	0.02%	0.02%	KERNELBA...		0	0x7ffa2fbb1755
[00007FFA2FBACADC]	0.03s	0.03s	0.01%	0.01%	KERNELBA...		0	0x7ffa2fbacadc
[00007FFA2FBCE11A]	0.03s	0.06s	0.01%	0.02%	KERNELBA...		0	0x7ffa2fbce11a
[00007FFA2FBE0D60]	0.03s	0.03s	0.01%	0.01%	KERNELBA...		0	0x7ffa2fbe0d60
rvOK	0.02s	0.10s	0.01%	0.04%	C_test_pro...	[unknown]	0	0x7ff7d94cf49e
[00007FFA2FBCE73C]	0.02s	0.02s	0.01%	0.01%	KERNELBA...		0	0x7ffa2fbce73c
resize_wbuf	0.02s	0.02s	0.01%	0.01%	C_test_pro...	[unknown]	0	0x7ff7d94c5233
[00007FFA2FBCE0BC]	0.02s	0.02s	0.01%	0.01%	KERNELBA...		0	0x7ffa2fbce0bc
__rshift_D2A	0.01s	0.01s	0.00%	0.00%	C_test_pro...	[unknown]	0	0x7ff7d94cdecd
RtlEnterCriticalSection	0.01s	0.01s	0.00%	0.00%	ntdll	[unknown]	0	0x7ffa30471694
__gdtoa	0.01s	0.02s	0.00%	0.01%	C_test_pro...	[unknown]	0	0x7ff7d94cc902
[00007FFA2FBE0DAF]	0.01s	0.01s	0.00%	0.00%	KERNELBA...		0	0x7ffa2fbe0daf
__lo0bits_D2A	0.01s	0.01s	0.00%	0.00%	C_test_pro...	[unknown]	0	0x7ff7d94ce203

Figure 9: Code Profiler (Very Sleepy) Results

11. Subsystem 2 – Legend

Refer to Figure 3 for more details regarding Subsystem 2 overview.

Terms	Definition
HTTP Request	A transmitted string that contains the relevant information for the handler function to operate
JSON LIDAR Scan	A portable method of storing data for network transmission such that it can be used on multiple platforms.
HTTP Request Handler	Handler module that extracts the relevant parameters from the Request and send the required line to the Dataset Reader
Dataset Reader	Reads the line from the dataset containing the request scan point data
Lidar Dataset	Database containing multiple scans each composed of a fixed number of scan-point representing a single lidar scan's distance and orientation
JSON Converter	Packages the dataset data into a JSON object suitable for transmission over the network
HTTP Response Sender	sends the packages JSON data over the network to the device.

Table 6: Subsystem 2 Legend

12. Subsystem 2 – Preliminary Processes

This portion describes the ***rationale*** and ***reason(s)*** behind all decisions within Subsystem 2.

The design flow for creating Subsystem 2 proceeded on 3 separate tracks, as after generating the initial implementation in C, it became necessary to port it to the respective embedded platforms. The first track was the programming of a lidar simulator server on Python, the second an ESP-32 platform that would be suitable for edge computing and the final would be the software and hardware development required for an optimised implementation upon the Kria board. Each of these development tracks required using their specific software tools for programming as well as debugging to ensure appropriate performance and reliability.

12.1 Emulating the LIDAR Sensor

As we chose to avoid the use of live data and instead use an existing dataset, it became necessary to create an interface for turning the raw database scan points into a stream of live data suitable for most SLAM algorithms, as well as simulating the innate latency that came with all real lidar sensors. To achieve this, we would need to identify a universal platform that would be suited for both the ESP32 and Kria board, as well as find a reliable way for the required scan point data to be broadcast by the server onto the devices. We created a simple server in Python that used the CSV file and HTTP libraries to broadcast the required data based on the appropriate HTTP request.

12.2 Implementation of Subsystem 1 into ESP32

ESP32 development was achieved using the ESP-IDF development environment for visual studio code that simplified the process of flashing as well as debugging the board. As our ESP32 came in a developmental board configuration it wasn’t necessary to create or solder any complicated wiring to get the debugging working. We had to design and code an interface that would read the data from the server and transmit it reliably to the edge processor that would be responsible for running the actual computations.

12.3 Implementation of Subsystem 2 into KV260

The Kria development proceeded on two separate tracks, one with regards to the actual software implementation that would be adapted from the existing code and a second one that involved the design and verification of the hardware co-processor that would speed up the processing on the embedded device to allow the system to deliver comparable performance to that of desktop platforms.

Development and hardware realisation makes use of the Co-processor development within the Xilinx family of tools, particularly Vitis and Vivado to program and verify the co-processor. Integration of the hardware and software was achieved through the PYNQ platform which allowed easy integration of Python/C code with hardware overlays containing the co-processor, with an online Jupyter notebook providing an easy-to-access remote GUI and development environment for working on the board. Actual compilation was handled on the terminal line using the GCC compiler directly.

13. Subsystem 2 – LIDAR Simulator Design Flow

Refer to Figure 3 for more information.

One of the key components for the program in our testing stage was a Python script that served to emulate the performance and behaviour of a real-world LIDAR sensor. The actual program serves the role of the HTTP server that formed an interface between our separate algorithm implementation and the actual dataset which would be stored as a CSV file on the same computer as the server which would then send a request for a specific scan point to the Server, which would read the file to the appropriate line and send the response with the required data if it existed, or else with an error message if it didn't. The creation of this server was a pre-requisite before development could begin on the ESP32 as the small memory capacity of the ESP32 required an external source be used to hold the dataset.

Our motivation for using such a high-level protocol was to ensure compatibility between the drastically different systems we tested our device on, with the ESP's native support for the HTTP protocol and suite of helper libraries being the deciding factors given its ordinary limitations.

13.1 Protocol Explanation

The various algorithm implementations of Subsystem 1 would send a request via their local area network to a server address that had been hard coded into their source code. The default port 8080 of the server would be open for the processing of these requests. A HTTP GET request formatted as shown below would then be sent from the SLAM system when it required a new scan of data.

Request: `http://192.168.137.1:8080?param=0001`

The actual processing and transport would be abstracted away using vendor-provided libraries, The response would then be sent as one long chain of floats, in a textual array that would be decomposed by the actual device. The position of each float in the array was the angle of the scan relative to the robot orientation which made it redundant to send angular information along with the chain of scan points.

Response: `[60.00,89.90,32.22,…]`

13.2 Development Process LIDAR Simulator

This was one of the fastest aspects of the program to develop, with verification being handled via curl commands on a desktop command line, and the actual server development being a simple modification from the demo code shown to introduce the server.

Due to limitations of time, we were unable to optimize the transmission protocol to directly transfer the float instead of converting it to text, but in calculations of latency, we found the performance impact minor (see later sections on performance) compared to the computational burden meaning that optimization in this arena would not have meaningfully affected the final system performance.

14. Subsystem 2 – Hardware Realisation of Subsystem 1 on ESP32

Note: Subsystem 1 is the algorithm integrated into edge computer while Subsystem 2 is integrated into the HTTP Server as shown below.

Figure 10 shows the hardware realisation on block diagram. There will be some fundamental hardware differences for ESP32 edge accelerated system as compared to that of KV260 FPGA accelerated system.

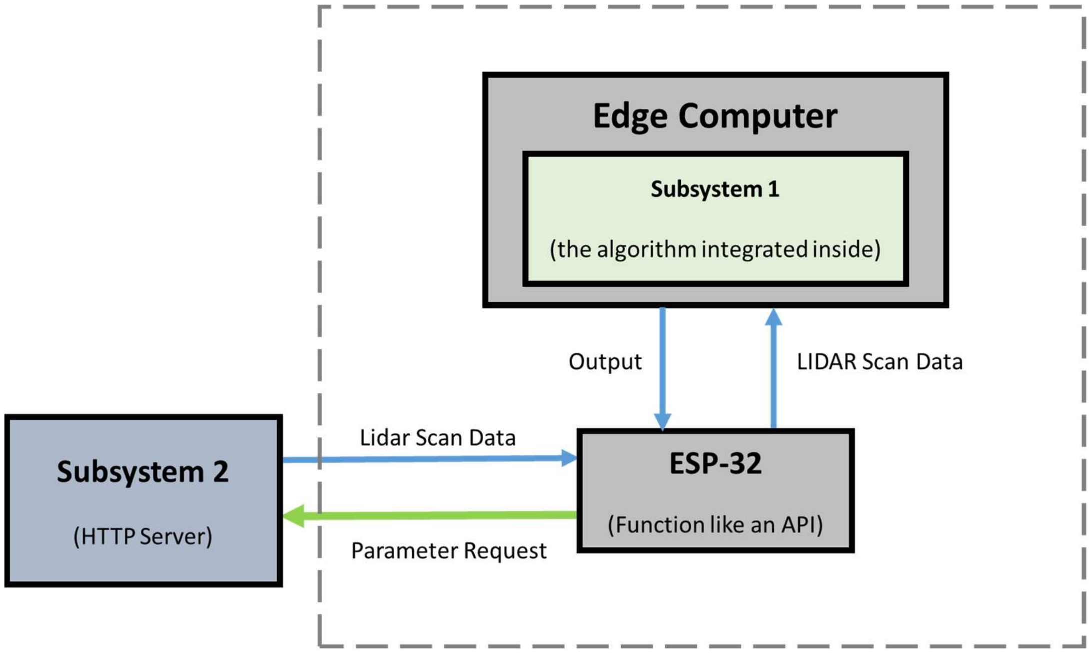


Figure 10: Hardware Realisation Block Diagram of ESP32

The ESP-32 development design flow faced initial challenges with finding a way to store and access the database file that was going to serve as a dataset for processing. One of the key limitations of the ESP-32 was its extremely limited memory capacity with the device having only 16 MB of flash memory. Hence, we would have to wire an external flash memory which would introduce additional latency and complexity, or else we would have to transmit individual points rather than the whole dataset.

14.1 Setting up the Development Environment

As previously mentioned, development was handled with the ESP-IDF which allowed a GUI to use the previously command-line-only toolchain used for Espressif development. Actual flashing was done via the UART port with the development board. A USB-OTG connection on the dev board allowed software to be flashed via the serial port while another offered an actual serial console for printing debug statements as well as monitoring the internal system states.

14.2 Challenges faced when integrating Algorithm into ESP32.

In terms of both saving development time and as well as processing efficiency, we found it useful to reuse the HTTP protocol previously described for transmitting scan data from the LIDAR emulator for use in the transmission of data to the edge processor. This was aided by the native support Espressif provided for wireless network connections. One of the challenges faced with development was the need for the Espressif to serve as both server and host; able to request data from the lidar server while also serving as the host for data requests from the lidar sensor; all while managing this on an extremely limited amount of memory with the functional RAM of the device limited to less than 52kb of RAM which made proper memory management, with the use of memory commands to free-up long used arrays essential.

Furthermore, initialization for a network connection and the ability to send a warning in case of a network disconnection was essential; with the primary commercial use of the Espressif chip being to integrate conventional embedded systems with limited network interfaces with wireless networks given the Espressif chips native support for both Bluetooth and Wi-Fi connections. One of the advantages of picking the esp32 is the wide array of demo and hobbyist programs with open-source code under an open license that we could adapt for our own purposes. Meaning that development could proceed much faster through re-use of open-source frameworks.

Another challenge was needing to familiarise with the basics of Real-Time Operating Systems (RTOS) and scheduling for a program that might need to multi-task between reading data and sending out responses. RTOS was the light-weight Operating System (OS) that the board came with native support. Setting priorities and the behaviour of the Espressif in handling the multiple tasks it had in play was another challenge in the systems development.

12.1.1 Code Integration Solution

It immediately became apparent from working with the ESP32 and reviewing its specification that running our planned algorithm on the actual chip itself would be impossible within any practical timeframe, hence we would have to rely on the use of edge computing for practical results. Once we accepted this design choice the primary challenge in the project was working with the network code to ensure both the integrity and throughput of the required data to the edge server.

14.2.1 Potential Implementation of ESP32's Built-In DSP Package

One of the potential avenues for exploration was the in-built DSP package that came equipped with the ESP32 that included a package for a Kalman filter which we briefly explored the possibility of using as a backup implementation that would allow the data to be processed solely on the ESP32 instead of edge computing. It quickly became apparent that our specific implementation would not be suitable for the digital processing offered due to it being used primarily for limited data points with just a few parameters rather than the map our algorithm was expected to handle.

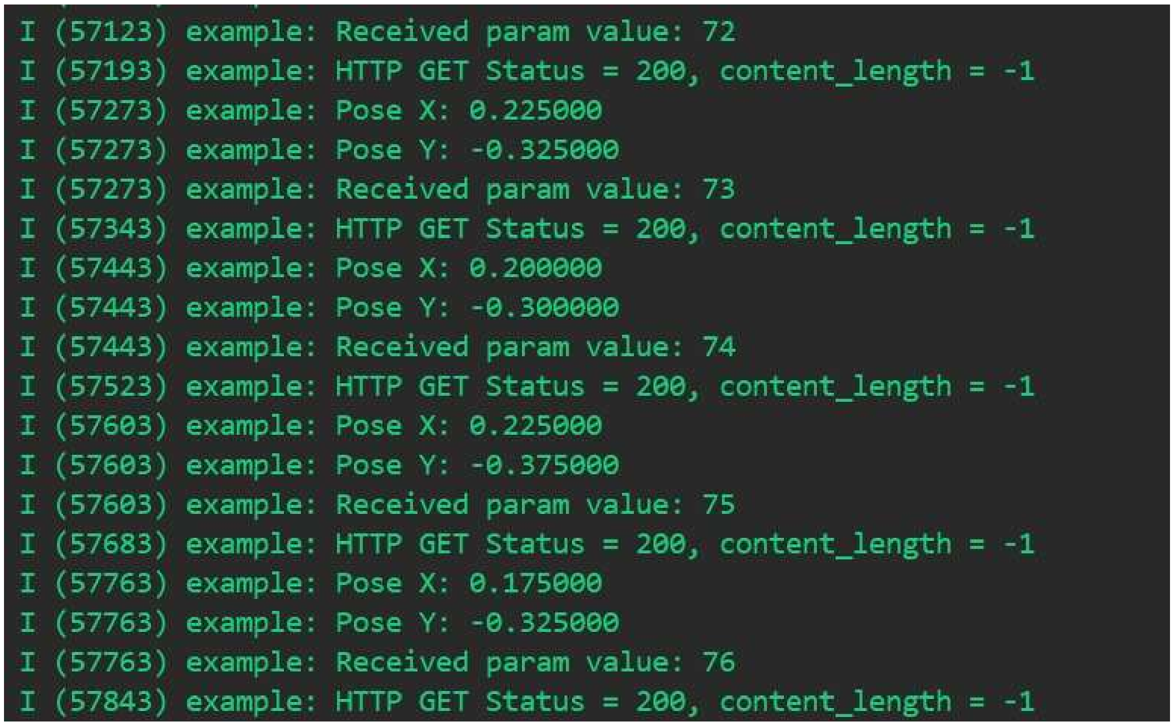
14.3 Throughput Challenges

Our initial plan was to transmit the scan-points to the ESP32 via the serial console, we attempted to use the UART protocol where the ESP32 could send requests for scan-point to an external computer which would respond with the appropriate data. The issue was that we found ourselves unable to set a sufficiently high-baud rate for the large number of scan points that would have to be transmitted. Furthermore, complications came with the fact that using the UART protocol as the main method of data transfer, rendered the internal state and behaviour of the Esspressif opaque due to a lack of human-readable serial output; drastically slowing down development and making it unsuited for further usage.

14.3.1 Throughput Solution

Our solution was to instead switch to using the Esspressif Wi-Fi-connection module connecting it to a local network on the same server as the edge computer and LIDAR simulator; with the Wi-Fi module capable of supporting sufficiently high throughput for our purposes.

14.4 Output at ESP32



```
I (57123) example: Received param value: 72
I (57193) example: HTTP GET Status = 200, content_length = -1
I (57273) example: Pose X: 0.225000
I (57273) example: Pose Y: -0.325000
I (57273) example: Received param value: 73
I (57343) example: HTTP GET Status = 200, content_length = -1
I (57443) example: Pose X: 0.200000
I (57443) example: Pose Y: -0.300000
I (57443) example: Received param value: 74
I (57523) example: HTTP GET Status = 200, content_length = -1
I (57603) example: Pose X: 0.225000
I (57603) example: Pose Y: -0.375000
I (57603) example: Received param value: 75
I (57683) example: HTTP GET Status = 200, content_length = -1
I (57763) example: Pose X: 0.175000
I (57763) example: Pose Y: -0.325000
I (57763) example: Received param value: 76
I (57843) example: HTTP GET Status = 200, content_length = -1
```

Figure 11: Output Values at ESP32

Figure 11 shows the output pose values when it is sent back to ESP32. This is essentially a proof of concept of the block diagram at Figure 10.

15. Subsystem 2 – Hardware Realisation of Subsystem 1 on KV260

Figure 12 shows the hardware realisation on block diagram. There will be some fundamental differences for the KV260 FPGA accelerated system as compared to ESP32 edge accelerated system as shown in Figure 10.

Note: Unlike in Figure 10, Subsystem 1 is broken up into Subsystem 1i and Subsystem 1ii.

According to Heading 10 above where profiling of the algorithm is done, accelerating the bottleneck at Euclidean distance transform is the key to improve the overall performance of the system. Refer to Heading 9.6 for more information of Euclidean Distance Transform Algorithm.

The Kria board proved to have the most complicated development cycle of all aspects of the project; with the 3 distinct stages of software, hardware, and integration each providing their challenges.

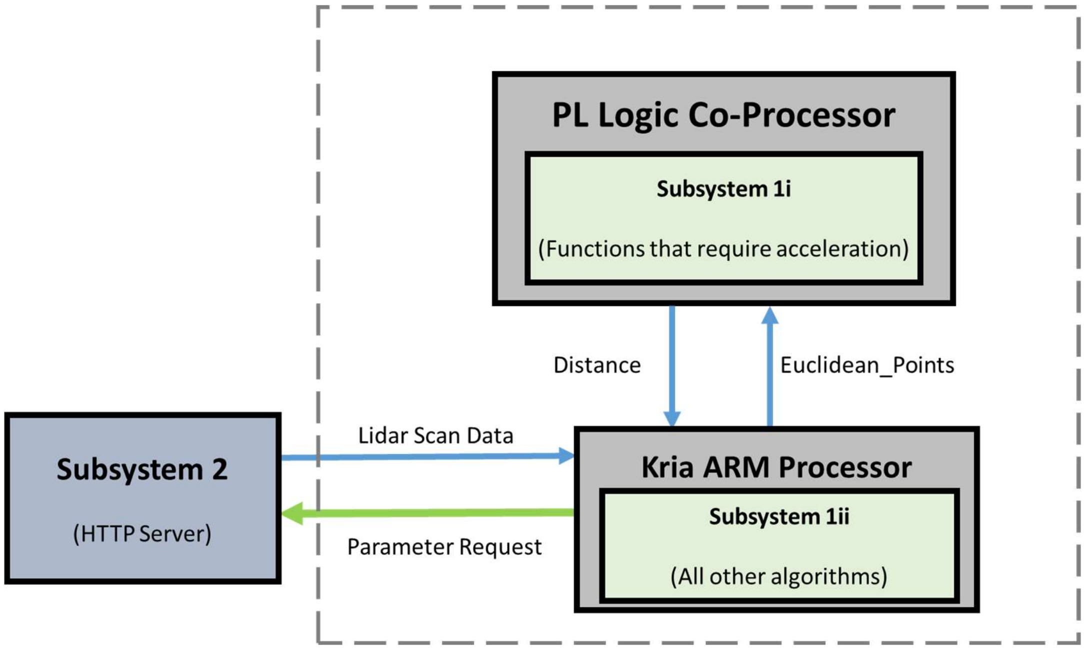


Figure 12: Hardware Realisation Block Diagram of KV260

15.1 Kria Board Setup

15.1.1 Kria Board Setup – Challenges

One of the first challenges we faced in the use of the Kria Board was deciding the operating system and development paradigm. The device was capable of bare-metal operations: suited for FPGA testing and verification, Peta Linux: suited for embedded development or Ubuntu: a Linux Desktop development. Though we attempted bare metal programming, we found the challenge of interfacing with networks far beyond our technical expertise and quickly installed Ubuntu which offered the use of a wide array of conventional Linux development environments as well as a wealth of support online.

This posed its own problems down the line; when our initial choice of SD-card failed which locked us into read-only mode, meaning that much of our unsaved work on the Kria board was lost with the whole device having to be factory reset to allow a reinstallation of the operating system; but other than that singular incident allowed us to move much faster with regards to porting and running of our desktop code onto the kria platform.

15.1.2 Kria Board Setup – Solution

However, despite choosing to use a desktop operating system on the kria board; we still aimed to implement a co-processor and avoid having to hook the board up to an external desktop and other peripherals. This was where Xilinx PYNQ’s framework offered the solution to our problems providing an easy method of integrating the PYNQ code while at the same time allowing us to work remotely through its Jupyter notebook that allowed direct editing of files remotely in a desktop browser.

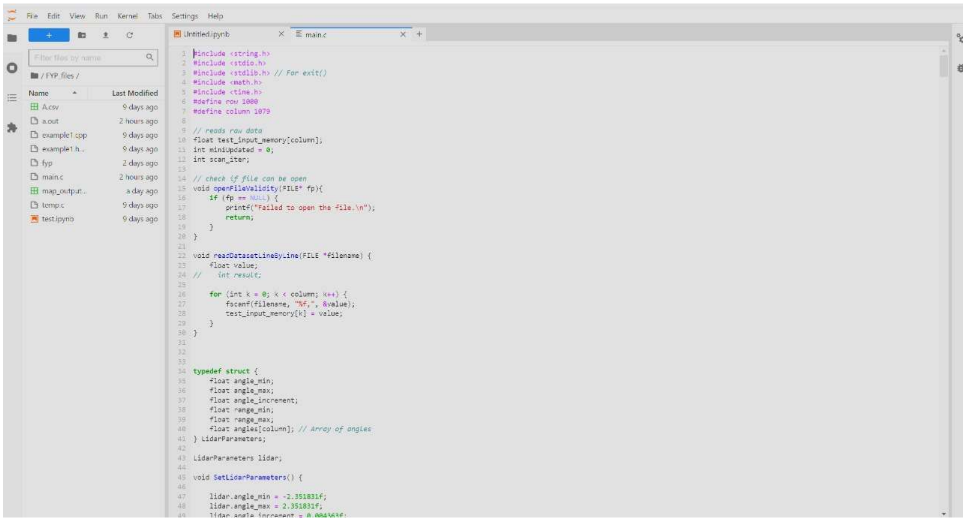
15.2 Kria KV260 Software Development

15.2.1 Xilinx PYNQ Framework

The Xilinx PYNQ framework is designed to simplify the integration of hardware and programmable logic components into software; by allowing the hardware complexity to be abstracted away into overlays which are generated from bitstreams which allows the hardware components of a program to be treated as a library file. It also comes with built-in integration with Jupyter notebooks allowing easier remote development on desktops and downloaded files, removing the need for special set-ups for flashing, integration or debugging.

Furthermore, the PYNQ platform ran Cpython which came with native support for Pybind11 which allowed direct implementation of native C code with the Jupiter notebook as well as integration with Python functions and importantly for our purposes, headers of co-processors; meaning that desktop code would need minimal changes to function with the co-processor allowing reuse and speeding up development time; by reducing the need to verify and test existing blocks of code.

15.2.2 Jupyter Notebook



Name	Last Modified
A.csv	9 days ago
a.out	2 hours ago
example1.cpp	9 days ago
example1.h...	9 days ago
fyp	2 days ago
main.c	2 hours ago
map_output...	a day ago
temp.c	9 days ago
test.ipynb	9 days ago

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h> // For exit()
#include <math.h>
#include <time.h>
#define row 1000
#define column 1079

// reads raw data
float test_input_memory[column];
int miniUpdated = 0;
int scan_iter;

// check if file can be open
void openFileValidity(FILE* fp){
    if (fp == NULL) {
        printf("Failed to open the file.\n");
        return;
    }
}

void readDatasetLineByLine(FILE *filename) {
    float value;
//    int result;

    for (int k = 0; k < column; k++) {
        fscanf(filename, "%f,", &value);
        test_input_memory[k] = value;
    }
}



typedef struct {
    float angle_min;
    float angle_max;
    float angle_increment;
    float range_min;
    float range_max;
    float angles[column]; // Array of angles
} LidarParameters;

LidarParameters lidar;

void SetLidarParameters() {

    lidar.angle_min = -2.351831f;
    lidar.angle_max = 2.351831f;
    lidar.angle_increment = 0.004363f;
```

Figure 13: Jupyter Notebook Framework Example

Jupyter Notebook is a framework designed to make code more portable and readable various notebooks to be accessed on a browser-hosted application that would be annotated with text and divided into segments to allow individual chunks to be run independently. For our purposes, the main advantage it provided us with was a remote development environment.

15.2.3 GCC

One of the issues faced in the initial stages of development with the Kria board was a lack of SDK that supported C which meant that both compilation and debugging would have to be handled directly on a command-line level. The open-source GCC compiler was the obvious choice coming natively with the Ubuntu install; however, shifting towards command-line development required becoming familiar with several tasks that had previously been abstracted away including manual linking of libraries, object name inclusions as well as version control.

15.2.4 Development Process on Kria

The actual development processes on the Kria board were straightforward due to the desktop OS allowing a generally straightforward reuse of the edge computation code from earlier; simply adapted to the Unix filesystem rather than requesting data from the ESP32.

One of the fundamental issues we faced porting the desktop software to Windows was the need to replace the curl library as the one we had chosen had no existing Linux implementation. We chose to focus on getting a pure software implementation of our code working on Ubuntu which would function identically to our initial Windows code which would read from a CSV file stored on the platform (which in the case of the KRIA board was easy to upload, the SD-card having more than enough storage to hold it). We also created a net-variant that could function integrated with the ESP32, serving as the edge processor of the ESP32 interfacing with it via curl commands; but found the challenge of integrating that with the hardware co-processor outside the scope far too time-consuming.

One example of the issue we faced in moving from Windows to ubuntu was the nonfunctional curl library that seemed to at random remove data without a clear pattern of operations due to an underlying difference in the way pointers functioned which required extensive research to uncover.

15.3 Kria KV260 Hardware Development

One of the unique features of the Kria KV260 board is a large amount of programmable logic suitable for transformation into digital circuits that can be used as custom co-processors. Unlike traditional processors which possess a wide array of instruction sets suitable for running general computations, digital circuits are capable of only one single fixed operation (or a limited range of them). This makes them suitable for accelerating applications whose performance is hindered by single functions or mathematically intensive operations that are suitable for parallelization.

15.3.1 Implementing a Co-Processor

The first step in designing a co-processor is to identify which specific software function is taking up substantial computational time while also being suitable for computations. The easiest way to do so is by running the software implementation through a profiler. The total time taken running a pure software implementation on the Kria board is ***<u>1303.32 seconds</u>*** for a reading of 1000 scan points. To examine which aspects of the program would lend themselves the easiest to hardware acceleration, it became necessary to profile the C-Execution to determine which functions served as the main cap on performance. From the profiling, it quickly became apparent that the `Euclidean_distance_transofFrorm_2` as shown in Figure 9 was a bottleneck we should accelerate.

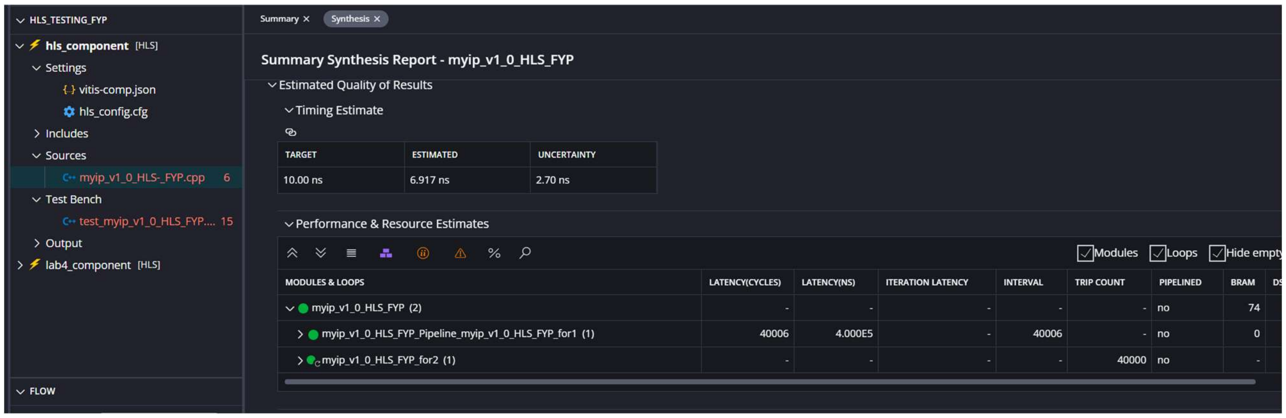
15.3.2 Accelerating Bottleneck of the Algorithm – Euclidean Distance Transform

From the profiler results as shown in Figure 9, we extract a processing time of ***<u>0.2279 seconds</u>*** $\left(\frac{227.9\, seconds}{1000\, Scans}\right)$ for the function per scan point in Euclidean Distance Transform. This is only a rough estimate that assumes the processing time is the same for all iterations which is not a safe assumption to make but can be assumed for our purposes given the lack of control-flow in the function, which we can use in the future to estimate what the performance improvement of the digital co-processor will offer over more conventional systems.

The purpose of this function is to be a distance transform function that takes in a binary map (one where every point is either a 1, representing occupied or a zero, representing unoccupied [16] and finds the distance between each point in the map to its nearest occupied map unless it's already occupied. It's computationally expensive due to the sheer number of comparisons and computations of binary distances that must be made.

Several aspects of the function suggest that it is suitable for hardware synthesis such as its structure as a four-level nested loop that can easily be unrolled and pipelined in a digital circuit to allow each to happen on the same cycle. Furthermore, the limited number of cycles (bound by an external definition) allows a predictable method for parallelising and running the operations in parallel with minimal data dependency between each iteration of the loop. All in all the function is well-suited for transformation into a digital circuit with large performance gains expected from designing

15.3.3 Implementing Co-Processor on Vitis HLS



Summary Synthesis Report - myip_v1_0_HLS_FYP

Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	6.917 ns	2.70 ns

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM
myip_v1_0_HLS_FYP (2)	-	-	-	-	-	no	74
myip_v1_0_HLS_FYP_Pipeline_myip_v1_0_HLS_FYP_for1 (1)	40006	4.000E5	-	40006	-	no	0
myip_v1_0_HLS_FYP_for2 (1)	-	-	-	-	40000	no	-

Figure 14: Vitis HLS Implementation.

Traditionally co-processors would be written directly in a hardware description language such as Verilog or VDHL in an architectural style, where the flow of the code would mimic that of the desired architecture of the circuit. Meaning that the circuit itself would have to be designed and specified before actual writing and testing could begin.

Recent advances however have allowed the rise of high-level synthesis where C-code with minor modifications, restrictions and a few hardware directives to indicate points of optimization can be directly transformed into hardware circuit designs suitable for flashing onto a FPGA as well as integration with other IP cores [17].

High-level synthesis allows Functions and Testbenches written in C to be directly converted into Hardware description language appropriate for implementation on the targeted platforms. Our choice of the Kria board meant we could use AMD/Xilinx design-suite which came with built-in HLS implementation in its Vitis tool suite that would be suitable for implementation design and then co-processors: without having to tedious manual translation and implementation. The design flow here was straightforward and involved taking certain functions from the software program that were suitable for hardware acceleration (suitable for parallelization) such as *Euclidean Transformation* copying the code over to the HLS tool. Modifying the function to handle a stream data format from the AXI-4 stream input.

It then involved verifying the functionality of the design by programming a testbench that would complete the same computations expected by the function in software and compare the results from each to verify the functionality. Part of the challenge here involved extracting intermediate variable values from our software implementation to have a suitable test case for us to verify the hardware functionality. Two levels of simulation had to be run; the first, a simple c-simulation was to ensure that the high-level Verilog pointed to a digital circuit capable of performing the same computations while the second involved simulating the produced simulation.

In addition to this raw HLS code, we experiment with a few directives to improve performance, including unrolling to allow more parallel processing as well as increasing the number of pipelines. The actual computations were being done in a series of nested-for loops that were suitable for both loop flattening and unrolling to drastically improve performance which is why I specified unrolling and other performance directives that would allow hardware to be duplicated to allow serial processing of the data.

Paragram **(ways to optimise)**	**Function**	**Target**
Array Partition	Partitions the input map data into multiple smaller chunks allowing more read and write accesses at each cycle	Input_MAP and OUTPUT_MAP
Loop Unroll	Creates multiple independent digital circuits for handling each iteration of a loop allowing far more parallel operations	Euclidean Computation Function
Loop Merge	Merge the input, processing and output loops into one operation prevent stalls and latency for operations	Overall Program

Table 7: Ways to Optimise HLS code

According to Figure 14, we can extract a performance of ***<u>0.0004 seconds latency</u>*** ($4.000e5\ ns$ on screenshot).

15.3.4 Verification

The C implementation is easily verified; with a randomly generated bitmap being pushed into the code and a software implementation running parallel to the hardware one to verify the accuracy. Complexity arises once we seize a netlist circuit and wish to test it; due to the large size of the data we are pushing (640 kbytes) meaning that far too many flip-flops and cycles must be emulated for it to be feasible to do a full verification on our desktop PC.

Instead, we run the simulation on a smaller map input (a 10x10 map) that is a subset of the full map the accelerator is expected to take in. This means our co-processor has to be designed with varying map-input sizes to allow the design verified with a smaller map to be used for the full-size data array.

15.3.4.1 Limitations of Hardware Design

With regards to our design, one of the limitations is that the whole system is blocked by the need to transform the map meaning that regardless of which method of interfacing is selected the software will be halted during the co-processor operations; with there being no obvious target for operations that could be done by the computer code during the operation due to the dependency of past outputs on future outputs.

15.3.4.2 Challenges – Choosing between AXI-FIFO and AXI-DMA

We are now faced with a design choice for our method of integrating the HLS IP with the existing embedded system. We can choose a FIFO input where all data-reading and more importantly writing is handled directly by the software which simplifies the hardware requirements but means the software must halt while the co-processor is being use.

Or instead, we can use the AXI-DMA to directly write the data into the RAM for the program to pick up, skipping the need for any data halt or losses during the interface section of the code at the cost of having to configure the DMA for direct memory operations.

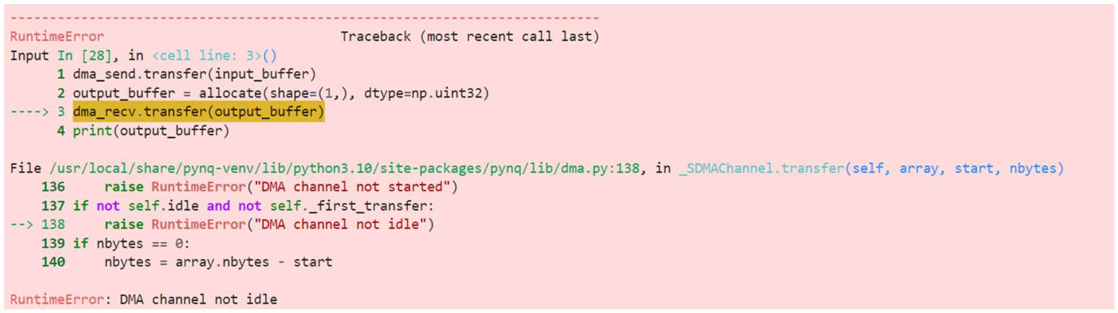
AXI-DMA is a good fit if we want to consider reducing latency as well as being better supported by the PYNQ framework we are using to integrate with our code.

15.3.4.3 Challenges – Implementing AXI-DMA

However, it is important to note that the Kria board is only available a few years ago (2021). HLS is also not very mature and unknown bugs are very common while using tools only available in recent years. Although it is beneficial to work with the latest technology, we must consider the fact that newer boards tend to be less well-documented than that of boards in the market for a substantial period. Kria board is after all still a relatively new board.

According to [18], a forum discussion post discussing issues of AXI-DMA that multiple people face, issues has been ongoing since the year 2023 and currently (2024), there are possibly more unknown bugs left unexplored.

The lack of well-documented issues faced made AXI-DMA difficult to implement within the timeframe. However, despite the challenges, we have implemented a HLS IP which can provide us with theoretical figures of merits.



```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [28], in <cell line: 3>()
      1 dma_send.transfer(input_buffer)
      2 output_buffer = allocate(shape=(1,), dtype=np.uint32)
----> 3 dma_recv.transfer(output_buffer)
      4 print(output_buffer)

File /usr/local/share/pynq-venv/lib/python3.10/site-packages/pynq/lib/dma.py:138, in _SDMAChannel.transfer(self, array, start, nbytes)
    136     raise RuntimeError("DMA channel not started")
    137 if not self.idle and not self._first_transfer:
--> 138     raise RuntimeError("DMA channel not idle")
    139 if nbytes == 0:
    140     nbytes = array.nbytes - start

RuntimeError: DMA channel not idle
```

Figure 15: HLS Encountered Bug (Example)

One of the errors that limited our ability to fully integrate our co-processor with the DMA was a glitch with regards to the AXI stream interface. The DMA kept asserting a busy signal despite the processing being done and a fixed output size being determined. According to a forum post this was a long running glitch of the program that required the assertation of a KEEP signal which is reserved for unaligned data transfers.

During the Capstone period, the forum showing a similar issue remained unresolved since 2021 [19].

15.3.5 Vivado Block Diagram

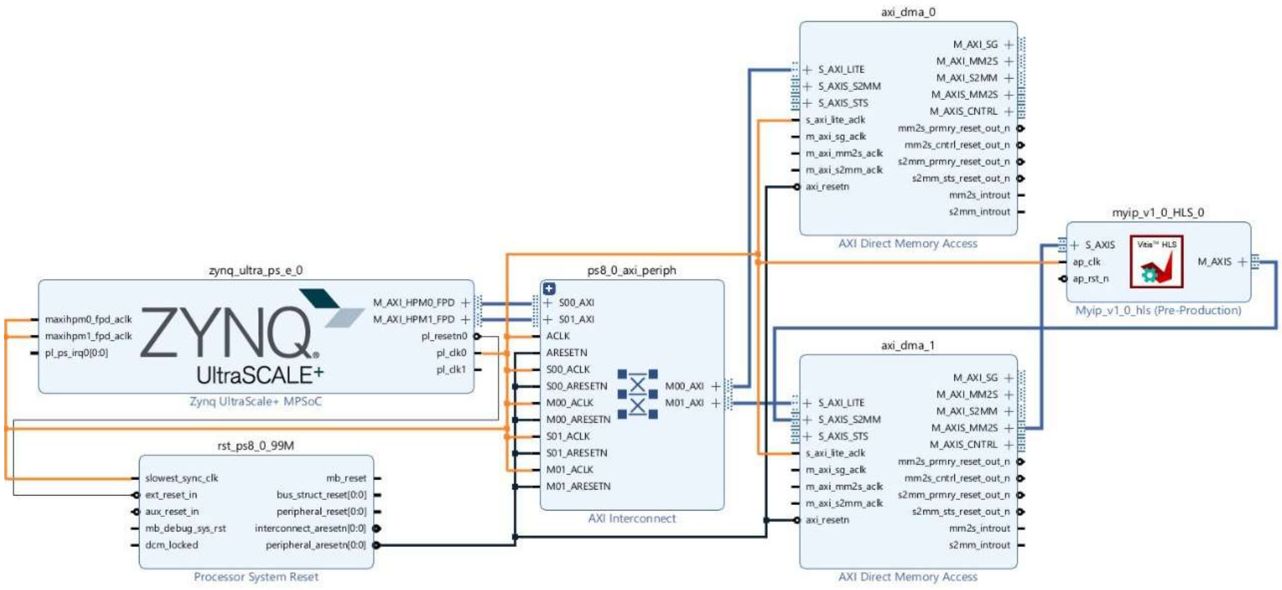
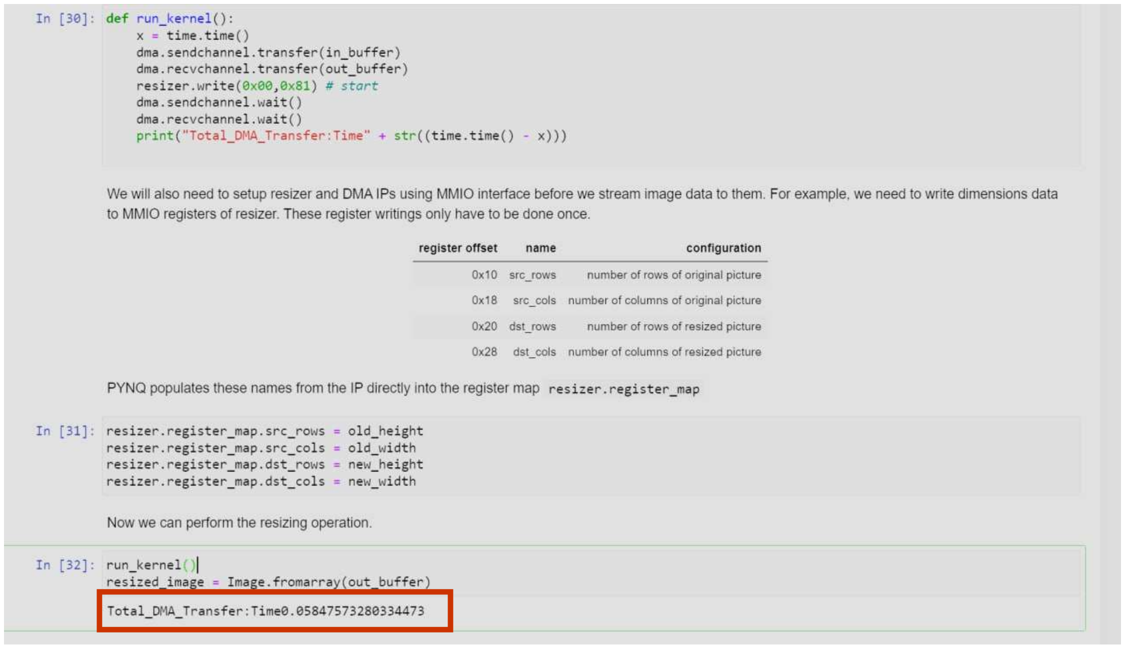


Figure 16: Vivado Block Diagram Design (AXI-DMA)

Once the co-processor functionality was verified it became necessary to integrate It with the Kria boards SoC, which is a ZYNQ Ultra-scale+ integrating it with the DMA module. The DMA module also must be configured appropriately as shown in Figure 16 to ensure that the data is written to the correct address for processing. The other modules are glue logic blocks to connect and interface the DMA with the SoC appropriately; the resulting hardware platform is then synthesised with the bitstream uploaded onto the board via the PYNQ interface before being imported and instantiated.



```
In [30]: def run_kernel():
             x = time.time()
             dma.sendchannel.transfer(in_buffer)
             dma.recvchannel.transfer(out_buffer)
             resizer.write(0x00,0x81) # start
             dma.sendchannel.wait()
             dma.recvchannel.wait()
             print("Total_DMA_Transfer:Time" + str((time.time() - x)))
```

We will also need to setup resizer and DMA IPs using MMIO interface before we stream image data to them. For example, we need to write dimensions data to MMIO registers of resizer. These register writings only have to be done once.

register offset	name	configuration
0x10	src_rows	number of rows of original picture
0x18	src_cols	number of columns of original picture
0x20	dst_rows	number of rows of resized picture
0x28	dst_cols	number of columns of resized picture

PYNQ populates these names from the IP directly into the register map `resizer.register_map`

```
In [31]: resizer.register_map.src_rows = old_height
         resizer.register_map.src_cols = old_width
         resizer.register_map.dst_rows = new_height
         resizer.register_map.dst_cols = new_width
```

Now we can perform the resizing operation.

```
In [32]: run_kernel()
         resized_image = Image.fromarray(out_buffer)

         Total_DMA_Transfer:Time0.05847573280334473
```

Figure 17: DMA Implementation on PYNQ

According to Figure 17, the total DMA hardware plus software is ***0.058 seconds*** for a set of data the size 3840×2160 (this is the default matrix input for PYNQ DMA demo), so the transfer rate per word is $7.0499\ ns$, which we then multiply by the size of each scan which is a 400×400 input matrix (the input matrix of Euclidean Distance transform2 function).

Overhead Calculations on DMA
$Total\ DMA\ Time\ (Overhead\ back\ and\ forth) = 0.058475\ s$
$No.of\ words = 3840 \times 2160$ → matrix comes default with PYNQ.
$Timing\ per\ word = \frac{0.058475}{3840 \times 2160} = 7.0499\ ns$
$No.of\ words\ for\ Euclidean\ Distance\ Transform2 = 400 \times 400$
$Overhead\ Time\ per\ scan = (7.0499 \times 10^{-9}) \times (400 \times 400) \approx 1.13\ ms$
We ultimately can round to ***0.001 per scan***.

Table 8: Overhead Calculations for DMA

15.4 Hardware Performance Measurements / Benchmarks

From our C/RTL simulation, we can benchmark an estimate for the performance of our co-processor by comparing the estimated performance of the co-processor (which is the worst time) with that of the actual function to create an estimated performance improvement from our hardware co-processor. First, we must calculate how much performance time is used for the Euclidean Distance Transform function that we aimed to optimize.

From Figure 9, we see that Euclidean distance Transform function exclusively used about ***89.98%*** of the computational run-time, with each scan point spending an average of ***0.2279*** seconds in this function during the overall life of the computer program. We can scale this up for the Kria board assuming that the functional elements scale equivalent. However, note that numbers and percentages may vary from time to time depending on different profilers and different processors.

Timing Calculations for Kria KV260

$Percentage\ of\ software\ time\ in\ Euclidean\ Distance\ Transform\ 2 = 89.98\%$

$Theoretical\ Total\ Time\ for\ Euclidean\ Distance\ Transform\ on\ Kria$

$= 89.98\% \times 1303.32$

$= 1164.72\ s$

$Additional\ overhead\ from\ interfacing$ (According to Table 8)

$= 0.001\ seconds$

$Latency\ of\ Hardware\ accelerated\ HLS\ implementation$ (According to Figure 14)

$= 4 \times 10^{5} \times 10^{-9}$

$= 0.0004\ s$

$Hardware\ accelerated\ time\ by\ Kria\ board$

$= 1000 \times (0.001 + 0.0004)$

$= 1.4\ s$

$Theoretical\ Total\ Time\ by\ Kria\ board\ (post\ hardware\ acceleration)$

$= 1303.32 - 1164.72 + 1.4$

$=$ ***140 s***

Note: 140 s timing is theoretical as AXI-DMA has certain platform bugs that are not very well-documented as mentioned in 15.3.4.3.

Table 9: Timing Calculations on Kria KV260

Software and Hardware Implementation Timings of Algorithm on Desktop vs Kria KV260		
Program	**Desktop**	**Kria KV260**
Total Software Timing	253.28 s (According to Sleepy Profiler at Figure 9)	1303.32 s
Euclidean_Distance_Transform2 (Software)	227.93 (According to Sleepy Profiler at Figure 9)	1164.72 s (According to Table 9)
Euclidean_Distance_Transform2 (Hardware)	NA	1.4 s (According to Table 9)
Hardware Accelerated	NA	140 s (According to Table 9)

Table 10: Software / Hardware Implementation Timing on Kria KV260

From our hardware coprocessor, we can extract an estimated processing time of ***0.0004 seconds*** from the C/RTL simulation. Assuming a conservative estimate of ***0.001 seconds per scan-point*** for interfacing we can estimate the actual co-processor computing time to be ***140 seconds***; with the time of the actual computation of the eudclidean_transform2 becoming irrelevant to the overall performance of the system.

16. Overall System – Potential Acceleration on KV260 Board

Since this board comes included with FPGA fabric, it means that we can potentially create a custom hardware accelerator to offload heavy computations such as the Euclidean Distance Transform Function onto the FPGA and send it back to the processor once calculation is done. After profiling the code to check for the bottleneck of the code, we could possibly accelerate said bottleneck by making use of a FPGA accelerator.

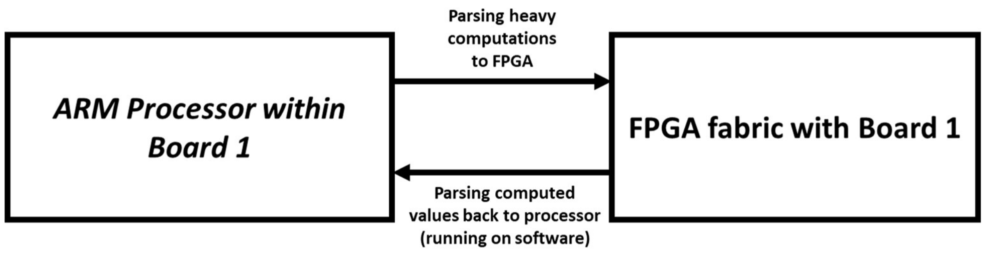

Parsing heavy computations to FPGA
ARM Processor within Board 1
FPGA fabric with Board 1
Parsing computed values back to processor (running on software)

Figure 18: Acceleration Method 1

16.1 Possible Advantages of KV260 Board

16.1.1 Lower Latency

FPGAs can possibly be lower latency when heavy computations are offloaded because these computations can be optimised for very specific tasks.

16.1.2 Energy Efficiency

Although FPGAs consume more power than that of ASICs, they are found to consume lower power than that of CPUs as shown in Figure 19. Conventionally, many LIDAR-SLAM algorithms are implemented on microprocessors and microcontrollers due to the advantage of flexibility in changing algorithms.

However, it is evident that LIDAR-SLAM calculations consume more power on CPUs as compared to FPGAs as shown in Figure 19.

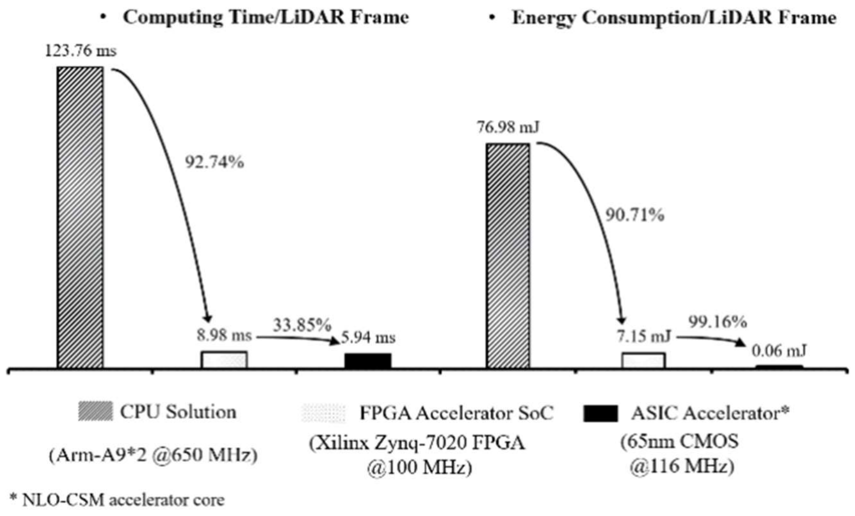


Figure 19: Comparison of the computing time and energy consumption per LiDAR frame among the CPU solution, FPGA hardware accelerator and ASIC accelerator of the NLO-CSM algorithm (Courtesy of [20])

16.1.3 Parallel Processing

Unlike CPUs which run algorithms sequentially, FPGAs can run its hardware in parallel. If we offload the computation to FPGAs, this can possibly increase throughput and reduce latency of the algorithm during runtime.

16.2 Possible Concerns of Acceleration on FPGAs

16.2.1 Large Overhead to and from FPGA

Overheads associated with passing the data to the coprocessor is inevitable. Using a coprocessor is worth it only when these said overheads + the co-processor's processing time is significantly less than the time taken for a pure software implementation.

16.2.2 Steep Learning Curve:

FPGAs are currently not very popular in robotics because hardware description languages (HDLs) used to run FPGAs have a steep learning curve. It takes a longer time to run the exact algorithm on FPGA as compared to running it on a CPU.

To capitalise on FPGA's advantage of parallelism and energy efficiency without taking a long time, we can profile the code to check for bottlenecks on the CPU. We will only accelerate the bottlenecks and not the entirety of the code.

17. Overall System – Potential Acceleration on ESP32

Since this board is meant for IoT and not meant for heavy computations such as LIDAR SLAM implementations, we can only compensate by running said heavy computations outside of ESP32. Running heavy computations on edge is a possible solution.

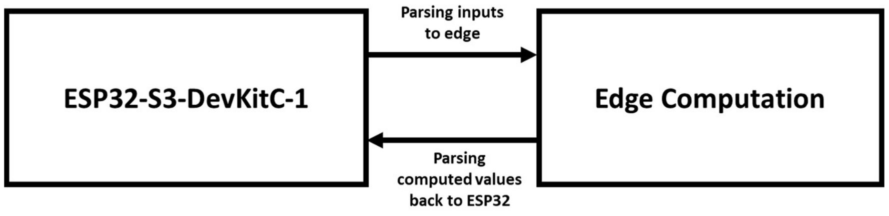


Figure 20: Acceleration Method 2

17.1 Possible Advantages of ESP32-S3-DevKitC-1

17.1.1 Lower latency

Edge computing can reduce the latency as compared to sending data to a remote data centre (E.g. Cloud computing). It can be useful if we want to have real-time responses.

17.1.2 Security

Companies that prefer to keep their computations confidential can run their code on edge. The processor (E.g. ESP32) is only responsible for parsing data as an API. As long as the data centre is protected, the security of the computations is most likely secure.

17.1.3 Scalability

Edge computing can distribute its computation resources across a range of devices allowing it to handle multiple devices.

17.2 Possible Concerns

17.2.1 Initial Cost

The cost of deploying and maintaining edge devices is expensive. Unless a company (E.g. NUS have their own servers) has a need for such edge devices like data centres, it might as well not be worth if it is a single-use case.

17.2.2 Network Connectivity area

Although edge can distribute is resources over a certain area, the connected devices still need to be within a certain vicinity to be able to connect. If the connected device moves out of the data Centre’s area of connectivity, connection might be lost.

18. Implementation and Actual Findings

For fair comparison, we will compare both boards with an equal number of LIDAR scans (1000 scans).

The total time taken is ***253.28 seconds*** for pure test calculations on a laptop (AMD Ryzen 7 6800U). We will be using the same laptop throughout to compare the calculations.

18.1 Software Implementation on KV260

Note: This Software Implementation does not include any implementation on FPGA yet for ease of debugging.

The total time taken is ***1303.32 seconds*** for pure software implementation on the Zynq ARM Cortex-A53 within the KV260 board (without FPGA acceleration).

18.1.1 Evaluation based on Software Implementation on KV260

It runs slower than expected on KV260. However, it is not surprising that it is so since the processor that the algorithm is running on is an in-order processor rather than an out-order processor.

18.1.2 In-order processor vs Out-of-order processor:

In-order processors run instructions in order while in the case of out-of-order processors, one instruction does not necessarily depend on the next. By running instruction stages ahead, it reduces latency. Since the ARM Cortex-A53 is an in-order processor, it could be a possible reason why the execution on KV260 is so slow.

Furthermore, according to [21], ARM Cortex-A53 is an 8-stage pipeline processor. 8 stages is considered relatively less as compared to conventional CPUs which can potentially affect the throughput.

18.2 Hardware Accelerated Implementation on KV260

Note: This hardware accelerated implementation is a theoretical value as AXI-DMA was not fully implemented due to lack of well-documented issues that made AXI-DMA difficult to implement (Refer to 15.3.4.3 for more information).

The total time taken is ***140 seconds*** if we accelerate the bottleneck function (Euclidean distance transform) on FPGA.

18.3 Implementation on Edge ESP32

Note: This Implementation on Edge is tested on a laptop (AMD Ryzen 7 6800U). As ESP32 does not have sufficient memory,

The total time taken is around ***302.89 seconds*** on the laptop or nearby edge. Data is transmitted via internet (http protocol).

18.4 Power Consumption Comparison

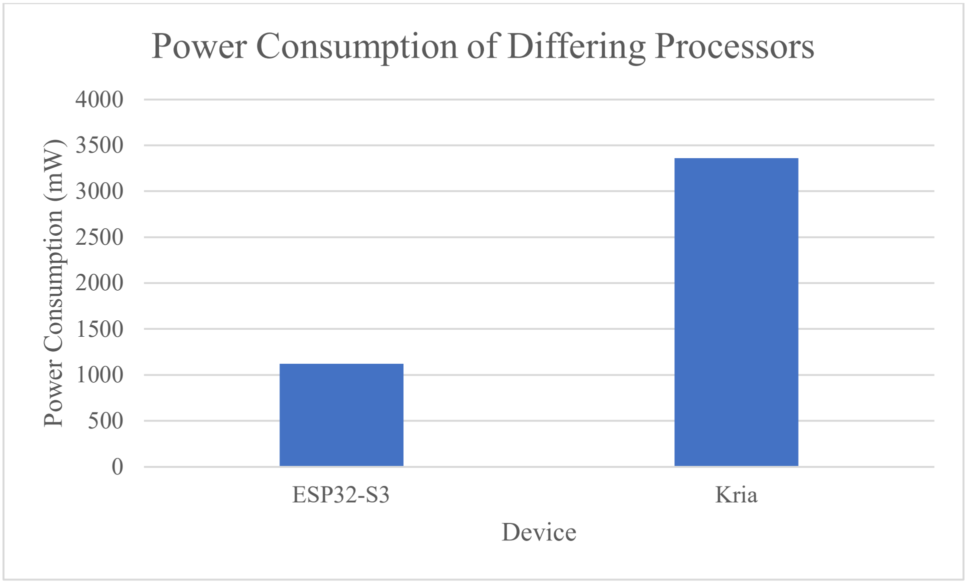
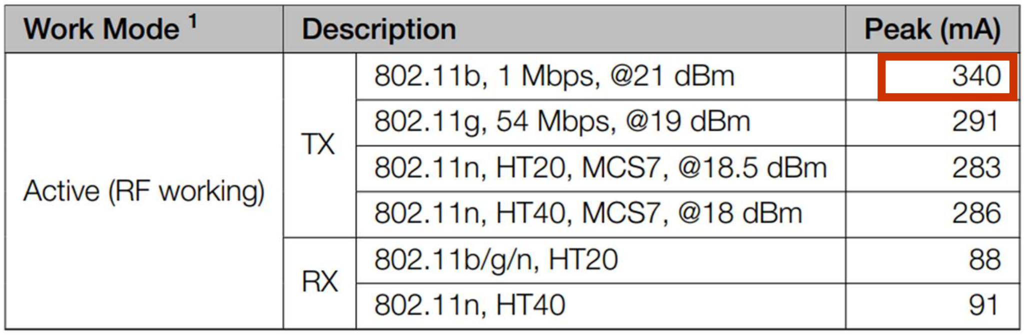


Figure 21: Power Consumption Comparison of ESP32 and Kria

18.4.1 Power Consumption – ESP32

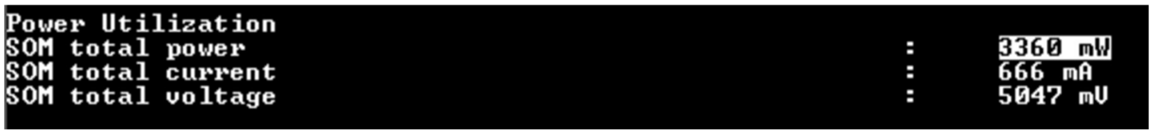


Work Mode [1]	Description		Peak (mA)
Active (RF working)	TX	802.11b, 1 Mbps, @21 dBm	340
		802.11g, 54 Mbps, @19 dBm	291
		802.11n, HT20, MCS7, @18.5 dBm	283
		802.11n, HT40, MCS7, @18 dBm	286
	RX	802.11b/g/n, HT20	88
		802.11n, HT40	91

Figure 22: Current Consumption of ESP32-S3 at Work Mode (Courtesy of [5])

The ESP32 run-time power consumption is hard to estimate due to the low power of the actual device making it unsuited for active power benchmarking. Instead, we can create an estimate of actual power consumption from the datasheet which shows that active mode power usage for the ESP32 is ***1122mW*** (which we can find by multiplying the current by the voltage input of 3.3V).

18.4.2 Power Consumption – Kria KV260



```
Power Utilization
SOM total power                                          :      3360 mW
SOM total current                                        :      666 mA
SOM total voltage                                        :      5047 mV
```

Figure 23: Power Consumption of Kria KV260

By using the built-in Xilinx performance monitor while the algorithm is running; we can estimate the total power consumption of the device, with the ***3360mW*** with a fair amount of the power consumption being overhead associated with running the CPU.

18.4.3 Edge Power Consumption:

Edge power consumption is irrelevant as it is meant to simulate a fixed base station connected to a mains supply which is not subject to the conventional limitations that embedded devices require. Further complicating measurement is the numerous background tasks that the operating system runs makes it hard to determine the specific power consumption of any single program.

19. Evaluating Trade-offs of Different Systems

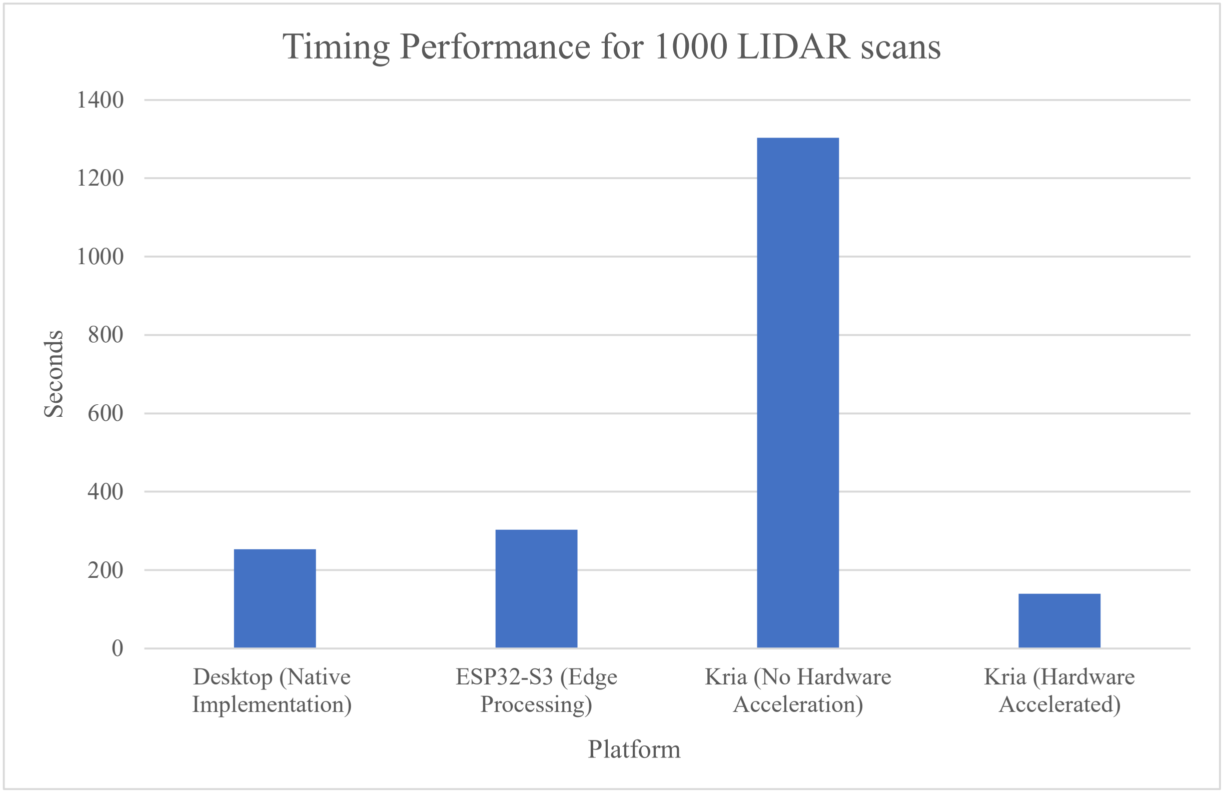


Figure 24: Timing Comparison of Different Implementations

Metrics	System 1	System 2
	Kria KV260 (ARM + FPGA)	**ESP32-S3-DevKitC-1 (Board 2) + Edge Acceleration**
Time Taken	**1303.32 seconds** for 1000 LIDAR scans **140 seconds** (theoretical timing if accelerated on FPGA)	**302.89 seconds** for 1000 LIDAR scans (with edge acceleration)
	System 1 is approximately 2 times faster than that of System 2 (theoretically). However, it is important to note that System 1's accelerated timing is theoretically calculated (practical values may vary depending on network environment).	
Cost	US $249 MSRP [7] ≈SGD $340.81	SGD $22.19 [9] → **excluding cost of Edge**
	Board 1 cost approximately 15 times more than that of Board 2	
Acceleration Method	Potential acceleration on FPGA Fabric (High-Level Synthesis)	Accelerated with Edge (A nearby computer/server)
Target Application	Designed for the purpose to accelerate locally	Primarily intended for IoT (Internet of Things)
Memory	4GB of RAM	52kBytes of RAM (excluding edge memory)
Power	**3360mW**	**1122mW** (excluding edge power consumption)
	System 1 uses approximately 3 times more power on average than that of System 2.	

Table 11: Updated Comparison Table of Different Systems

19.1 Cost

Cost depends on many factors, and it differs from situation to situation. Depending on various circumstances, different volumes, and different scenarios, it may cause the cost to change. These are the various considerations and depending on the different parameters in different circumstances, the cost might increase or decrease.

19.1.1 Cost in Low Volumes

In low volumes, system 1 may possibly have some advantages over system 2. This is because system 2 requires a nearby edge such as a laptop or a local server which is an additional cost to factor in. System 1 itself (S$340.81) can function on its own without additional hardware. Hence, in a theoretical scenario, system 1 would be more suitable for low volumes as compared to system 2.

However, if we consider factors such as maintenance of system 1 or system 2 in real life, it could possibly drive the cost up further as well.

12.1.3 Cost in High Volumes

In higher volumes, system 2 may possibly have some advantages over system 1. This is because sending calculations to a nearby edge can incorporate more than one processor. This means that system 2 has good scalability as compared to system 1 which operate its calculations locally (Board 1 itself is essentially an independent system).

Furthermore, higher volumes of boards bought in bulk can further reduce the cost. Board 2 is already 15 times cheaper than that of Board 1 if buy either one of them individually. If bought in bulk will further reduce the overall cost.

However, if we factor in miscellaneous costs like maintenance, cooling and rental expenses then the calculations became far more complicated.

19.2 Latency

System 2 (302.89 seconds) has a lower latency than that of System 1 (1303.32 seconds) for 1000 LIDAR scans computation (~4.3MBytes). However, it is important to note that System 2 incorporates a nearby edge (AMD Ryzen 7 6800U is used to test).

System 1 incorporates board 1 which is an 8-stage pipelined and in-order processor as mentioned in 18.1.2. This could be a possible reason why system 1 is slower than that of system 2 for calculating 1000 scans since the incorporated processor used for calculations in System 2 is a state-of-the-art, out-of-order processor.

However, the timing is not definite and “different edges” in System 2 give very different latency. In this case, latency for system 2 is lower than that of system 1. System 2’s timing could possibly increase or decrease depending on the processor used.

Sending data to calculate at a nearby edge requires internet. Depending on the speed of the internet, it can yield different results (E.g. 4G vs 5G internet).

The number of LIDAR scan inputs is also a possible factor that can affect latency. For example, if we are only required to measure a small map, the total LIDAR scans will reduce which causes the overall latency to reduce.

19.3 Memory

It is possible to implement very heavy computations like LIDAR-SLAM locally on a processor in Board 1 (ARM Cortex A-53 has 4GB RAM). For 1000 LIDAR scans, it takes 1303.32 seconds to calculate. However, the trade-off of algorithms calculated locally in system 1 requires more memory.

It is very difficult to implement very heavy computations like LIDAR-SLAM calculations on Board 2 (52kB RAM) within System 2. However, System 2 incorporates a nearby edge which can significantly reduce memory requirements on local processors like Board 2. The larger memory requirements are incorporated on the local server. This means that an extensive amount of memory is still required but does not have be located locally and can be placed on the edge processor.

19.3.1 Troubleshooting

System 1 is an independent system on its own. Scalability wise, System 2 can possibly incorporate multiple processors at the same time. For larger volumes, it is possible that troubleshooting can be minimised because any error in algorithmic calculations can be done on edge rather than troubleshooting locally on each processor.

However, for situations that require only a small volume of processors, system 1 can possibly be easier to troubleshoot. For example, if only one processor is needed to measure a relatively small environment, system 1 itself is sufficient.

20. Conclusion

There are various ways to accelerate calculations on different systems (E.g. accelerate on edge, accelerate on FPGA, etc). However, it is important to consider various trade-offs before deciding on a particular acceleration method.

When evaluating the performance of both the Kria KV260 and ESP32-S3 for the mapping application we draw some broad conclusions and recommendations regarding future similar projects as well as more projects aimed at turning our results in a practical direction. Neither approach can be concluded as categorically superior but there is a definite skew towards practical applications of the hardware acceleration approach due to its vast performance improvement as well as reduced reliance on possibly unstable connections and avoiding the need for an edge computer to initiate the processing.

Certain use cases such as robots that must operate in isolated environments or use cases where servers are not readily available are simply not suitable for edge processing and hence it would be beneficial to use other hardware acceleration techniques for their mapping purposes. Yet in other situations such as robots that have to operate with extreme power constraints and low cost, make it such that edge processing is more suitable, provided a server is readily available.

References

[1] AMD, "Kria KV260 Vision AI Starter Kit," 2024. [Online]. Available: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html. [Accessed 10 March 2024].

[2] "ESP32-S3-DevKitC-1 v1.1 - ESP32-S3 — ESP-IDF Programming Guide Latest Documentation," Esspresssif, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html. [Accessed 5 11 2023].

[3] "ESP32-S3-DEVKITC-1 Price," DigiKey, [Online]. Available: https://www.digikey.sg/en/products/detail/espressif-systems/ESP32-S3-DEVKITC-1-N8R2/15199627.

[4] "ESP-IDF Programming GuideLogo," Esspressif, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ulp.html. [Accessed March 2024].

[5] "ESP32-S3 Series Datasheet," [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf. [Accessed 28 October 2023].

[6] "Kria KV260 Vision AI Starter Kit Data," 15 March 2022. [Online]. Available: https://media.digikey.com/pdf/Data%20Sheets/Xilinx%20PDFs/Kria_KV260_Vision_AI_Starter_Kit.pdf. [Accessed 28 October 2023].

[7] W. Hess, D. Kohler, H. Rapp and D. Andor, "Real-time Loop Closure in 2D LIDAR SLAM," 9 June 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7487258. [Accessed 28 October 2023].

[8] J. LV, "Scan Matching and SLAM for Mobile Robot in Indoor Environment," [Online]. Available: https://www.eng.hokudai.ac.jp/e3/alumni/files/abstract/d206.pdf. [Accessed 28 October 2023].

[9] "2D Cartographer Backpack – Deutsches Museum," Cartographer ROS, [Online]. Available: https://google-cartographer-ros.readthedocs.io/en/latest/data.html. [Accessed 5 November 2023].

[10] "Estimate Robot Pose with Scan Matching - MATLAB & Simulink," MATLAB, [Online]. Available: https://www.mathworks.com/help/nav/ug/estimate-robot-pose-with-scan-matching.html. [Accessed 28 October 2023].

[11] K. Sugiura and H. Matsutani, "A Unified Accelerator Design for LiDAR SLAM Algorithms for low-end FPGAs," 23 November 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9609886. [Accessed 28 October 2023].

[12] C. P. Sesmero, S. V. Lorente and M. D. Castro, "Graph SLAM Built over Point Clouds Matching for Robot," MDPI, 7 August 2021.

[13] meyiao, "LaserSLAM," GitHub, [Online]. Available: https://github.com/meyiao/LaserSLAM. [Accessed 28 October 2023].

[14] "Distance Transform of Binary Image - MATLAB Bwdist," MATLAB, [Online]. Available: https://www.mathworks.com/help/images/ref/bwdist.html. [Accessed 28 October 2023].

[15] R. Mitton, "codersnotes," 19 August 2021. [Online]. Available: http://www.codersnotes.com/sleepy/. [Accessed 25 March 2024].

[16] C. R. Maurer, R. Qi and V. Raghavan, "A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions," 2003. [Online]. Available: https://ieeexplore.ieee.org/document/1177156/authors#authors. [Accessed 24 March 2024].

[17] G. D. Micheli, "High-Level Synthesis of Digital Circuits," 1993, pp. 207-283.

[18] "AMD PYNQ Forum Discussion," October 2023. [Online]. Available: https://discuss.pynq.io/t/dma-stuck-when-using-custom-hls-ip/6247. [Accessed 2 April 2024].

[19] "AMD PYNQ Forum," Septermber 2021. [Online]. Available: https://discuss.pynq.io/t/runtimeerror-dma-channel-not-idle/3037. [Accessed April 2024].

[20] Q. Wang, A. Hu, D. Han, Y. Yu, G. Yu, Y. Li and C. Wang, "Hardware Accelerator Design of Non-linear Optimization Correlative Scan Matching Algorithm in 2D LiDAR SLAM for Mobile Robots," 24 April 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10103802. [Accessed 2023 October 2023].

[21] "Cortex-A53," ARM, [Online]. Available: https://developer.arm.com/Processors/Cortex-A53. [Accessed 19 March 2024].

[22] Espressif, "ESP32 Series Datasheet," 2024. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. [Accessed 2 April 2024].

Appendix: Health and Safety Criteria

Our project involved minimal work in the lab with the main risk being working with developmental boards as well as repetitive stress injuries that could develop while programming or writing for long periods of time. Developmental boards are essential tools for prototyping and testing electronic circuits and systems. While they are valuable for innovation and experimentation, it is crucial to observe specific health and safety criteria to ensure the well-being of individuals using these boards. This appendix outlines guidelines for promoting a safe working environment for working with the project.

1. Electrical Safety:

- Before working with developmental boards, ensure that they are powered off and disconnected from the power source.
- Use insulated tools and equipment when making connections or modifications to the board to prevent electrical shock.
- Verify that the power supply and voltage levels are compatible with the specifications of the developmental board to avoid damaging the components.

2. ESD (Electrostatic Discharge) Protection:

- Implement ESD protection measures to prevent damage to sensitive electronic components on the developmental board.
- Use anti-static wrist straps or mats when handling the board to dissipate any static charge and minimize the risk of ESD-induced failures.

3. Mechanical Safety:

- Handle developmental boards with care to avoid physical damage to components, connectors, and circuit traces.
- Avoid applying excessive force or pressure when inserting or removing components from the board to prevent mechanical failures.

4. Thermal Safety:

- Monitor the temperature of the developmental board and surrounding components during operation to prevent overheating.
- Ensure adequate ventilation and airflow around the board to dissipate heat effectively and maintain optimal operating conditions.

5. Workspace Organization:

- Maintain a clean and organized workspace free from clutter to minimize the risk of accidents and injuries.
- Store developmental boards and associated components in designated areas when not in use to prevent damage and facilitate easy access.

6. Prevention of Repetitive Stress Injuries (RSIs):

- Recognize the potential for RSIs, which can result from prolonged or repetitive tasks such as those associated with report writing or typing.
- Take regular breaks and rotation of tasks to minimize prolonged exposure to repetitive motions.