Capstone Proposal

Designing an Energy-Efficient Embedded Deep Learning-SLAM for Unmanned Ground Vehicles

Students:
Umer Bin Liaqat (ubl203)
Moaaz Assali (ma5679)
Fatema Alzaabi (fya210)

Capstone Mentor:
Professor Muhammad Shafique

Date: 2023/05/08

Table of Contents

1. Abstract

One major problem with using machine learning algorithms for autonomous robots is that they may not be optimized enough to be deployed on resource-constrained devices in terms of energy consumption and computational speed. This is especially true for the Visual Simultaneous Localization and Mapping (VSLAM) algorithms that perform computations on RGBD input. The proposed solution for this capstone is a Deep Learning (DL) VSLAM algorithm that is equipped with an optimization framework that performs optimizations such as buffer tuning, quantization, image-downscaling and temporal optimization. The algorithm is designed to be energy-efficient and computationally-inexpensive to allow for implementation in resource-constrained environments while maintaining a respectable degree of mapping accuracy. The finalized DL-VSLAM algorithm will be implemented on the real-world Unmanned Ground Vehicle (UGV), Clearpath Husky, for demonstration and thorough validation after being developed, trained, optimized, and tested.

2. Problem Definition:

2.1 Problem Analysis

In order to analyze the problem, we begin by asking the following questions:

1. Who has the problem?
2. What does the problem seem to be?
3. What are the resources available to solve the problem?
4. Where does the problem occur?
5. Why does the problem occur?
6. How does the problem occur?
7. What is “resource-constrained”?

After consulting with literature and our prior knowledge, we have determined the following answers to these questions:

1. Communities/ people who use autonomous/unmanned vehicles; systems utilizing autonomous movement
2. Such DL-VSLAM algorithms require highly complex computations leading to gigantic processing, memory, and energy costs. The ever-increasing complexity of DNNs aggravate the energy-efficiency challenges.
3. In terms of software, there is a wealth of open source algorithms and technologies that we can leverage. Chief among these are machine learning algorithms that can help us solve complex problems.
4. As autonomous vehicles are still a developing field of research, improvements are needed.
5. Visual features can be expensive to implement and develop.

6. Energy-Efficient Embedded Deep Learning-VSLAM for Unmanned Ground Vehicles are expensive to implement and thus come with higher priced vehicles.
7. Possessing limited CPU/GPU computation capabilities, using antiquated technologies and/or operation in remote settings for extended periods of time where robot energy conservation is paramount

One of the most prevalent problems that arises from the discourse above is that energy-efficient optimization used for deep learning-VSLAM features are hard to develop, and they require additional hardware which is expensive both in terms of monetary value and energy consumption. The use of machine learning algorithms in autonomous vehicles can be an energy-intensive process, as these algorithms typically require a lot of computational power to make decisions and control the vehicle. To make machine learning more energy efficient for autonomous vehicles, there are a few different approaches that can be used. One approach is to use more efficient machine learning algorithms, such as decision trees or random forests, which can make predictions and decisions using fewer computations than other types of algorithms. By using these algorithms, it may be possible to reduce the amount of energy required to run the machine learning system.

Another approach is to use hardware acceleration, such as graphical processing units (GPUs) or specialized machine learning chips, to accelerate the computational processes used in machine learning. These hardware components can perform many of the calculations required for machine learning much more quickly and efficiently than a traditional central processing unit (CPU), which can help to reduce the energy consumption of the system. Additionally, it may be possible to optimize the use of machine learning in autonomous vehicles by only running the algorithms when they are needed, rather than constantly. For example, the vehicle could use sensors to detect when it is in a complex or uncertain situation, and could then activate the machine learning algorithms to make decisions and control the vehicle. This could reduce the overall energy consumption of the system, while still ensuring that the vehicle is able to operate safely and effectively.

We will address this problem by leveraging the large wealth of machine learning algorithms and open source technologies. These would allow us to deliver superb results without the need for industry-level software development cycles. In addition, recent breakthroughs in machine learning have shown that it is possible to extract a wealth of information from simple two-dimensional images from normal optical cameras.

2.2 Problem Clarification

The key question that this capstone aims at addressing is, if and how can we map complex deep learning visual simultaneous localization and mapping (DL-VSLAM) algorithms on resource-constrained embedded computing platforms (such as, low cost Nvidia Jetson embedded GPUs, and Xilinx FPGA-MPSoCs) of autonomous systems, and demonstrating the developed concepts for a real-world use case of a UGV to demonstrate a full-system prototype. Figure 1 shows a *black box* abstract model representation of our initial model. In this representation, the inputs are the RGB images from the monocular camera, and our model uses the information provided by the cameras to provide the appropriate pose and map to output. The process of extracting the relevant information from the RGB input and processing it to produce the map and pose is done by the deep learning and machine learning algorithms like DROID SLAM. These algorithms pass through the optimization framework which will enable for their implementation on resource-constrained environments.

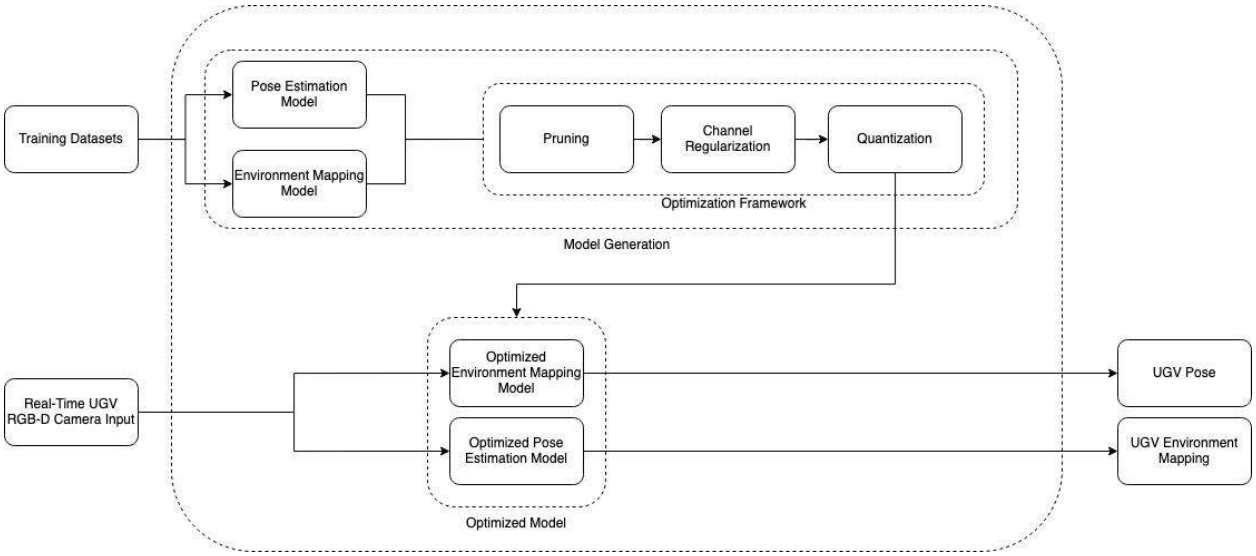


Figure 1. Representation of our initial model using a black box model

The main challenge of this work is designing, developing and implementing the processes inside the black box which would extract the necessary information from the feeds that will allow our model to make the relevant decision. Distance estimation using images from a monocular camera can be done using a machine learning model. Indeed, most of the processes we have mentioned will be machine learning algorithms, as they are highly suitable in our use case where we need to extract a large volume of information simply from a set of monocular camera images.

2.3 Problem Statement

Deep Learning (DL) Simultaneous Localization And Mapping (SLAM) algorithms have proliferated in advanced autonomous systems like autonomous vehicles, unmanned ground and aerial vehicles (UGVs, UAVs), and other types of mobile robots. These robots are used for remote mapping and exploration in harsh environments where human access is limited. Robot power conservation is important in such tasks especially when deployed for extended periods of time. Currently, there is a lack of robust yet computationally-cheap and energy-efficient DL-Visual SLAM algorithms. This capstone project aims to ameliorate this issue by developing an optimized DL-VSLAM algorithm which combines durability with cost-effectiveness and energy-efficiency that can be deployed on UGVs performing exploration and mapping tasks.

3. Design Constraints

3.1 Technical Constraints

3.1.1 GPU Memory

Deep learning models load the entire model on the GPU to optimize for speed and parallelizability, which means that we will also be limited by the available GPU memory. We will be using the Husky's on-board computer, Jetson TX2, for testing our models. The model is intended to run on systems with similar specifications, so the same 8GB GPU memory of the Jetson TX2 GPU will be the memory constraint for our models.

3.1.2 Camera Input

The models will all be trained using a single RGB video and depth data from a depth sensor (this is also commonly referred to as monocular RGB-D camera). The RGB-D input will be limited to one source only and no other RGB-D data sources will be used. The camera used will be Intel® RealSense with a 1920x1080 resolution and 30fps. The input resolution to the model and the fps will be changed to match the resource requirement for the model to operate according to the required budget in terms of memory, energy, latency and accuracy.

3.1.3 3D Environment Mapping

The output of the model must be a 3D map of the surroundings that can be utilized by other non-mapping UGVs for navigation over uneven terrain. The information from 2D maps does not include height estimation which is a very important component for environment modeling.

3.2 Non-Technical Constraints

3.2.1 Different UGVs frameworks

The model should be modular and use libraries that can also be deployed on the different respective systems that UGV robots operate with. Additionally, the model should be compatible

with ROS and have a mechanism of digital simulation for testing and debugging purposes, such as being executable with UGV packages on Gazebo.

4. Criteria for Design Evaluation and Testing

The testing criteria is summarized in Table 1 below and explained individually afterwards.

Accuracy	**Active Energy consumption**	**Latency**
baseline of 3cm ATC, 75% mapping accuracy	10W expected for Jetson TX2 GPU	minimum 33ms/30fps to fully utilize depth info

Table 1: Testing criteria

4.1 Accuracy

The accuracy of the model refers to how close the depth estimation of the model is to the ground truth with the *actual* depth values. The model will be tested against many datasets such as RGB-D dataset and NYUDepth to assess the accuracy of our model. It will also be compared to other state-of-the-art models that tackle the same problem along with their respective trade-offs in the other evaluation metrics. The accuracy for pose estimation will be evaluated using the Absolute Trajectory Error (ATE) in centimeters as described in [9]. The expected baseline accuracy for pose estimation is 5cm. For mapping, the accuracy will be calculated by counting the percentage of pixels where the depth estimation is within 25% of the ground truth. For this definition of mapping accuracy, our expected baseline accuracy is 75%.

4.2 Latency

The latency of the model is essential for DL-VSLAM deployment on UGV robots because given the limited computational resources available, we still have to ensure that the model is able to produce depth estimations fast enough. This will be measured in frames per second (fps) and will be compared to other open-source models designed for systems with limited computational resources. The expected minimum latency that the model should be designed for is 33ms (or 30fps) in order to fully utilize the depth information from the RGB-D camera.

4.3 Power Usage

UGV robots are also limited by their battery capacity and cooling capabilities. Therefore, another important consideration is the energy efficiency of the model. This will be measured in watts using active power consumption of the GPU while running the model. The expected power consumption is 10W while the DL-VSLAM model is running on Jetson TX2 GPU.

5. Deliverables Statement

The main deliverable of this capstone project is to develop an energy-efficient embedded deep-learning visual VSLAM algorithm for unmanned ground vehicles.

By the end of Capstone II, we intend to have a fully functional Visual DL-SLAM algorithm implemented on the Clearpath Robotics Husky shown in Figure 2. If the implementation is successful and if the optimizations are efficient, this project has a strong potential for a product that will enable several innovative practical & real-world use cases for autonomous systems like reconnaissance, surveillance, etc. and would be a mandatory component in many UGV, Robots, etc. A github repository with proper documentation will be prepared for the optimized model's final code, streamlining the reusability of our model.

Figure 2: Clearpath Husky UGV

6. Conceptualization

Recently, the Deep Learning (DL) algorithms have proliferated in advanced autonomous systems like autonomous vehicles, unmanned ground and aerial vehicles (UGVs, UAVs), and other types of mobile robots, robotics for surveillance and reconnaissance. One of the key computational problems for these autonomous systems is simultaneous localization and mapping (SLAMs), which requires constructing/updating a map of an unknown environment and at the same time keeping track of the robot's location within this map. According to the recent surveys, due to the growing trends of autonomous vehicles, UAVs/UGVs, and mobile devices, the expected market of the SLAM will be about 25 billion USD by 2025 [1]. This has also motivated leading software/AI companies (such as, Google, Amazon, Microsoft, Qualcomm, Amazon, etc.) to work on this topic. The advanced SLAM algorithms leverage modern deep neural networks (DNNs) for achieving high accuracy, for instance, leveraging the Convolutional Neural Networks (CNNs) for interest point computation and the Graph Neural Networks (GNNs) for establishing

correspondence between input samples. On the one hand, due to these complex DNNs, such DL-SLAM algorithms require highly complex computations leading to gigantic processing, memory, and energy costs [2]. The ever-increasing complexity of DNNs aggravate the energy-efficiency challenges. On the other hand, these autonomous systems are subjected to stringent energy and resource constraints due to the limited compute and memory resources of their embedded hardware devices and restricted battery capacity. Moreover, these embedded computing platforms should be able to handle different types of DNNs such as CNN and GNN, as discussed above.

From existing literature, we can find several DL-VSLAM algorithms that are always improving in terms of accuracy. However, this improvement typically comes at the expense of a much more complex network with a higher power consumption. This includes all the networks outlined below like Active Neural SLAM, DROID-SLAM, and DeepFactors[3][10][11]. The existing literature is lacking in terms of a highly efficient DL-VSLAM algorithm for deployment in resource-constrained embedded systems. However, existing literature contains extensive research on model optimization and edge AI. Frameworks like NetAdapt and MorphNet provide a comprehensive approach to optimize pre-trained systems for low energy consumption. Additionally, common techniques such as quantization are well documented and built-in in major machine learning libraries such as PyTorch for direct implementation. Our project bridges the gap between very high power consumption of DL-VSLAM models and embedded systems that, on the other hand, require highly efficient models through the study and implementation of state-of-the-art optimization techniques and frameworks.

Morphological Chart:

Our framework consists of a single sub-module with multiple options for consideration as seen from the back box explained above. The table below shows the design options for each of the sub-modules in the framework.

Module	**Design Options**		
SLAM Algorithm (Mapping and Pose Estimation)	Active Neural	DeepFactors	DROID SLAM

Table 2. Morphological Chart

Pugh Chart:

A Pugh chart for the SLAM algorithm was made where the criteria for our sub-module selection were memory overhead, accuracy, latency and energy efficiency. Weights were assigned to the sub-modules while considering their importance to our final objective of making an energy-efficient DL-SLAM for UGVs.

SLAM algorithm sub-module refers to the existing SLAM algorithms that we have tested on our GPU 4x RTX A6000. Energy and Accuracy of the algorithms were given the heaviest weights since they form the most-essential criteria for our design evaluation as mentioned in Section 4. Memory is not a big concern for us at the moment since our Jetson TX2 GPU has 8GB of memory which would be quite sufficient for our optimized algorithm. Latency was given a weight of 2 (lower than energy and accuracy) because although a full utilization of 30fps from the RGB-D camera is ideal, the UGV can still function well with a higher latency (lower fps) and generate a sufficiently accurate 3D map. After conducting theoretical comparative analysis on the results and using one of the algorithms as the base comparison algorithm, each algorithm was assigned either (+/-) 1 or 2 based on its comparative performance.

		Design Options		
Criteria	**Weight**	**Active Neural**	**DeepFactors**	**DROID SLAM**
Memory	1	base	-1	-2
Latency	2	base	-1	-2
Energy	3	base	-1	-2
Accuracy	3	base	+1	+2
Total		base	-3	-5

Active Neural model as base

		Design Options		
Criteria	**Weight**	**Active Neural**	**DeepFactors**	**DROID SLAM**
Memory	1	+1	base	-1
Latency	2	+1	base	-1
Energy	3	+1	base	-1
Accuracy	3	-1	base	+2
Total		+3	base	-1

DeepFactors model as base

		Design Options		
Criteria	**Weight**	**Active Neural**	**DeepFactors**	**DROID SLAM**
Memory	1	+2	+1	base
Latency	2	+2	+1	base
Energy	3	+2	+1	base
Accuracy	3	-2	-2	base
Total		+5	+1	base

DROID SLAM model as base

It is clear from the pugh charts that Active Neural SLAM has the highest score and performs better for all criterias except for its accuracy where DROID SLAM is the most accurate. The ideal implementation will be that of Active Neural SLAM. However, it will become clear later that it is unable to satisfy our 3D mapping constraint. Although DROID SLAM's memory footprint and energy consumption are comparatively higher, there is a lot of room for potential optimization to make it more efficient. This makes DROID SLAM ideal for implementation as the eventual optimizations will take advantage of its high accuracy while matching the other algorithms in terms of energy efficiency and memory footprint. DROID SLAM's most significant disadvantage is its memory footprint which is around ~20GB, much greater than the available GPU memory of 8GB on Jetson TX2. Therefore, the main focus of optimizations will be to bring down the memory consumption of DROID SLAM to within the allowed domains.

7. Modeling, Simulation, and Experimental Plan

7.1 Modeling

The modeling of our DL-SLAM system starts with the selection of the base pose estimation and mapping model. One by one, each algorithm was executed in its raw state for further comparative analysis. The results from comparative analysis would decide which algorithm is ideal for optimizations.

The initial base model was Active Neural SLAM. Not only does it provide the best result in terms of the criteria needed, but it is also a more recent model that builds upon previous learning models and employs classical SLAM methods in addition to deep learning [3]. It also constantly re-analyzes its estimated pose and map built using short-term goals through local policies by minimizing errors in a specific window [3]. Global policies are also utilized in long-term goals by loop closure (i.e. detecting errors when the device returns back to the original supposed position) and re-visiting known landmarks in the map [3]. Despite the many upsides to Active Neural SLAM, its implementation was abandoned at the later stages due to its major limitation of producing only a 2D map output.

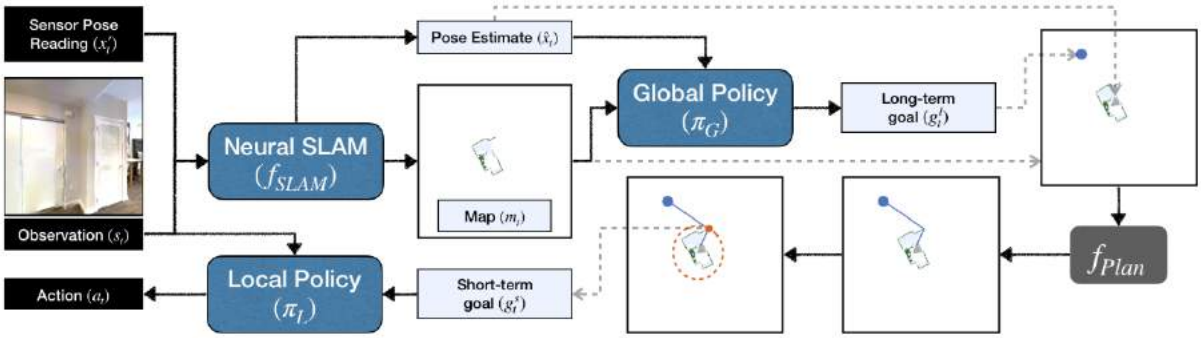


Figure 3: Full approach of Neural Active SLAM model and its sub-modules that will be used [3]

The second base model was DROID SLAM. It stands for Differentiable Recurrent Optimization-Inspired Design (DROID) and consists of recurrent iterative updates of camera pose and pixelwise depth through a Dense Bundle Adjustment layer. Its design is novel and uses an end-to-end differentiable architecture that combines the strengths of both classical approaches and deep networks. The DBA layer leverages geometric constraints, improves accuracy and robustness, and enables a monocular system to handle stereo or RGB-D input without retraining [12]. A simple representation of DROID SLAM working is shown in Figure 4.

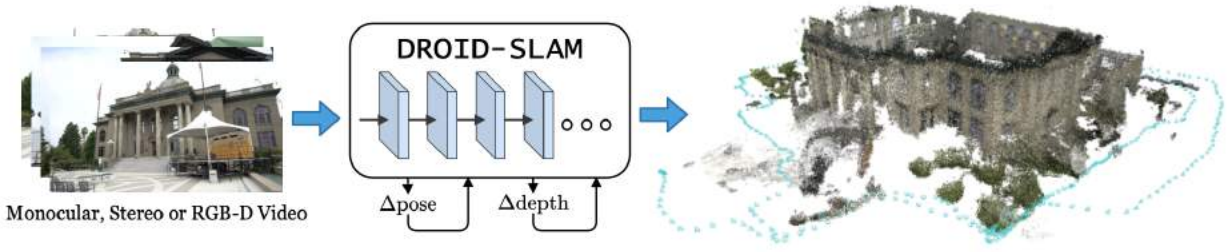


Figure 4: Simple representation of DROID SLAM's working

The third base model is NetAdapt. Despite its many advantages, its execution is very complicated. It will become clearer later that several complications made its implementation unfeasible.

Given the base model, we are presented with multiple optimization techniques such as pruning, quantization and buffer tuning. The main focus will be on the pruning aspect as that allows us to specify our budget for all of latency and energy and minimize accordingly. NetAdapt offers a complete framework for such a process and works by doing an exhaustive search through the entire network to meet the specified budget requirements while maximizing the accuracy [4]. It looks at specific filters and entire layers that contribute the most towards our budget needs and tries to maximize the accuracy given the constraints provided [4]. It also fine-tunes the network by running through the dataset both locally and globally to further optimize it [4].

To further optimize the network for resource-constrained devices, FX Graph Mode Quantization from PyTorch will be used to reduce the precision of the weights of the network and reduce memory usage along with computational load. Finally, MorphNet will be applied to the optimized model to fine-tune the network through channel regularization. This will help us tune hyper-parameters in the network to achieve optimal performance while still maintaining the same budget constraint. MorphNet works by automating the process of shrinking and expanding the network through a sparsifying regularizers across different layers and optimizes its strength to maximize performance [5].

7.2 Simulations

Given the structure of the model used, we can simulate all the criteria parameters, such as energy, latency and accuracy on robot simulation softwares like Gazebo and Unity. It is challenging to perform a large number of tests on the robot itself throughout the optimization process, which is why simulations in Gazebo will be used to constantly test the model. The Clearpath Husky comes with official support for simulation in Gazebo in the form of a robot package that replicates the original robot in movement, dimension and function. However, this doesn't eliminate the need for real-life tests, which will be done occasionally after the models simulated in Gazebo meet the expectations.

To generate the diverse environments, lighting, and weather conditions needed for the simulations, we will also need to simulate them in a 3D game engine like Unity. Together, Unity and Gazebo, will fully allow us to test the UGV robot sensors and movement along with the model's performance in a wide variety of conditions. These simulated environments will also be used to further train the model on environments that are not present in publicly available datasets.

The model's testing parameters will be constantly monitored. The memory footprint is measured by looking at the NVIDIA GPU memory under use when the testing is on-going. The energy-usage is more-or-less directly proportional to the GPU memory consumption. The accuracy is measured differently for each algorithm. Active Neural SLAM has an inbuilt configuration for the exploration task where accuracy is defined in percentage as area_successfully_explored/total_area. On the other hand, DROID SLAM's accuracy is measured through the Absolute Trajectory Error (ATE) which directly measures the difference between points of the true and the estimated trajectory.

7.3 Experimental Plan:

- Step 1: Implementation of the Active Neural SLAM on Gibson dataset

The default dataset available from Active Neural SLAM official github repository was used and the exploration task was completed. From our tests, the model was able to explore the specified map with 95% accuracy. Figure 5 below shows a sample run inside a house along with the generated map and a trace of the robot's pose. The pale light blue color corresponds to the explored area, whereas the gray color is for unexplored regions, and the green color represents obstacles found.

Figure 5: Sample generated map and pose trace for the initial model

We also tested the initial model on the RGB-D dataset of NYU Depth dataset V2 [8] and obtained the following results when running it on the workstation with 2x RTX 3090 GPUs. Note that this is not representative of the performance on the final target hardware, but serves as a metric for other optimization tests done on the same workstation.

Sequence	RMSE	Latency (ms)	FPS
360	0.066249	2.720	367.6
desk	0.017715	2.436	410.5
desk2	0.026611	2.449	408.3
floor	0.022268	2.456	407.1
plant	0.016654	2.397	417.1
room	0.043037	2.784	359.1
rpy	0.023389	2.348	425.8
teddy	0.035754	2.631	380.0

Table 3: Active Neural SLAM testing parameter results

- *Step 2: Gazebo/Unity simulation and creating custom environments (DROID SLAM)*

First, a proper interface with Gazebo/Unity and the Robot Operating System (ROS) was created with the appropriate nodes and topics. Then a custom Gazebo environment with various objects was created where the robot could navigate the environment and utilize its sensors. For further rigorous testing, a default 3D HDRP Unity simulation environment was chosen that closely replicated real-life lightning, objects and uneven terrain. Testing the model in real-life on the UGV robot will then be a matter of deploying the ROS modules onto the robot itself. Following that, we then tested the model with custom environments.

Gazebo:

In order to simulate the Clearpath Husky robot using ROS1 Noetic and ROS# packages, we first installed and configured the necessary dependencies. This included common ROS packages for the Clearpath Husky, which are usable for both simulation and real robot operation, such as husky_control, husky_description, husky_msgs, and husky_navigation.

Additionally, we needed to install desktop ROS packages for the Clearpath Husky, which may pull in graphical dependencies, such as husky_viz for visualization configuration and bringup. Finally, we installed simulator ROS packages for the Clearpath Husky, including husky_gazebo for Gazebo plugin definitions and extensions to the robot URDF.

Once the necessary packages were installed, we set up Husky's network and created a Husky workspace. From there, we launched Gazebo to begin the simulation as shown in Figure 6..

Gazebo is a common simulation tool used in ROS1 Noetic and is capable of simulating Husky's dynamics, including wheel slippage, skidding, and inertia.

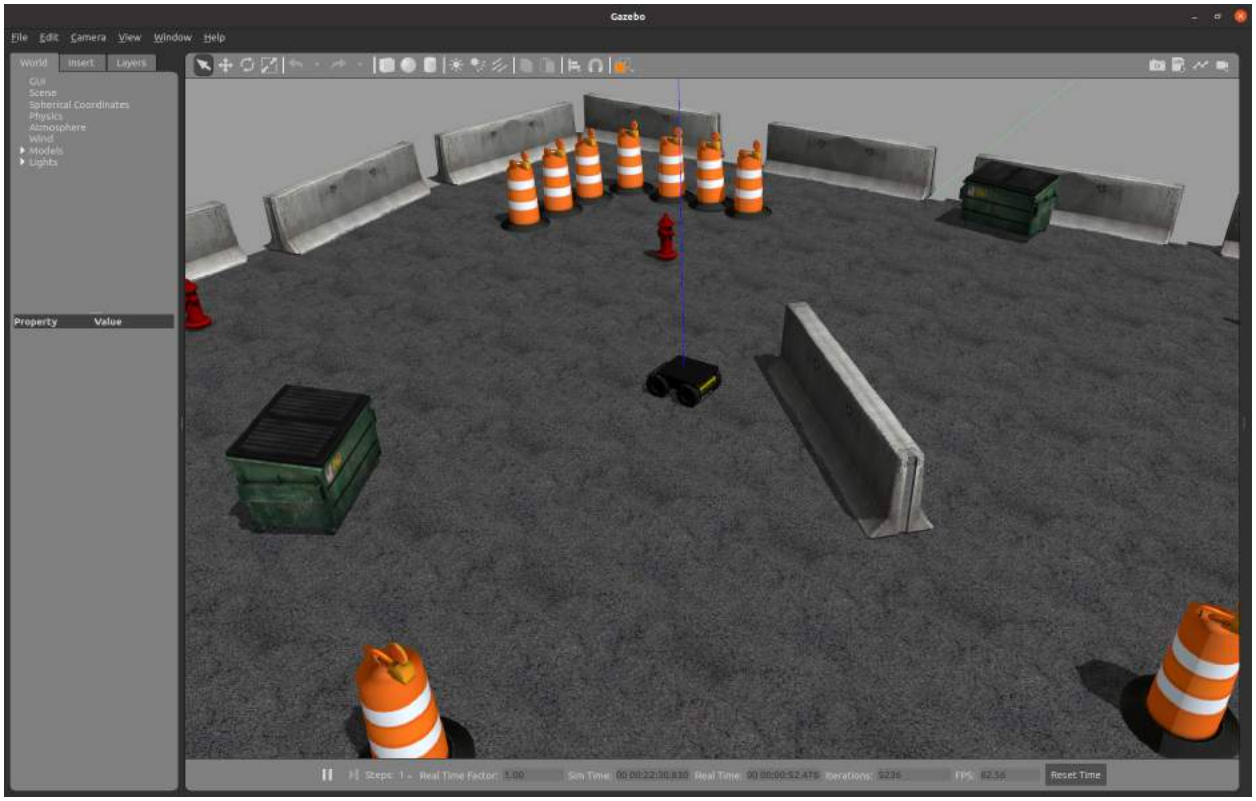


Figure 6: Gazebo custom environment with Husky

The Gazebo Client window provided a view of the "true" state of the simulated world in which the robot existed. It communicated with the Gazebo Server on the backend, which was responsible for maintaining the simulated world.

To view the robot's perception of its world, we launched rviz. Although rviz may look similar to Gazebo, it serves a different purpose. Rviz displays the robot's perception of its world, whether real or simulated. While Gazebo won't be used with a real Husky, rviz can be used with both real and simulated Husky.

Unity:

We initially attempted to simulate the Husky robot in Gazebo, but found that the graphics and physics simulation were not as realistic as expected. Therefore, we recreated our simulation in Unity, which provided more realistic graphics and physics simulation.

To achieve this, we integrated ROS with Unity using the ROS# package and developed a custom C# script to convert the Unity camera view to a ROS image message.

To begin, we created a new Unity project and imported the ROS# package. We added a camera to the scene and added the ROS# component to a new GameObject. We then configured the ROS# component to connect to the ROS master and advertise a new topic for the camera image.

Next, we created a custom C# script to handle the camera image conversion and publishing to the ROS topic. In the script, we created a RenderTexture object with the dimensions of the camera view and set the targetTexture property of the camera to the RenderTexture object. We then rendered the camera view and used a Texture2D object to read the pixel data from the RenderTexture object. After reading the pixel data, we converted the Texture2D object to a byte array using its EncodeToPNG() method. We then used the ROS# API to publish the byte array to the ROS topic created in step 2.

Finally, we built the Unity project and ran the simulation. The Unity camera view was streamed to the ROS image topic and could be visualized in RViz. The integration of ROS with Unity allowed us to simulate the Husky robot and enabled the implementation of the Droid SLAM algorithm.

Figure 7: Sample images from Unity environment in real-time

Figure 8: Generated 3d map and pose estimates (red squares) from the real-time unity image stream

- *Step 3: Testing the optimization techniques on the model and integrating custom environments*

The various optimization techniques will be applied to the working model. Each will be analyzed for its impact on the model following the outlined criteria (accuracy, energy, and latency). The custom environments developed for Gazebo simulation can also be integrated throughout the re-training that occurs within the optimization step. Step 2 and step 3 will go hand-in-hand and ensure continuous testing and development as the model is further optimized.

- *Step 4: Final extensive testing with UGV robot in real environment*

After obtaining the final model which has been tested with Gazebo simulations, and the intermediate models that were occasionally tested in real environments, the final model will be deployed on the UGV robot. It will undergo extensive testing in real-world environments with different terrains and conditions.

8. Final Design Expected

8.1 Optimized SLAM Model

After running the selected SLAM algorithms, except DeepFactors which proved unable to be executed due to technical complications, DROID SLAM was selected as our base model for further optimizations. Although Active Neural SLAM had better performance metrics, it produced a 2D map output. A 2D map is incompatible with our initial design constraints of the project. Despite our efforts, the 2D output could not be not scaled to 3D due to the immense complexity of the code.

8.2 Final Design Review

Figure 9 shows the final comprehensive design of the project. Starting with datasets (including custom generated environments), we train a full DL-SLAM model based on DROID SLAM for pose estimation and 3D mapping. The model will then pass through the image downscaling, temporal optimization, quantization and buffer tuning optimization techniques that further optimize it.

With the sub-models available at real-time, we can now provide the budget constraint and the real-time RGB-D camera input. We will then look up for the optimal sub-model given the budget

constraint and infer the UGV 3D pose (x, y, z along with yawn, roll, pitch) and a 3D map of the surroundings.

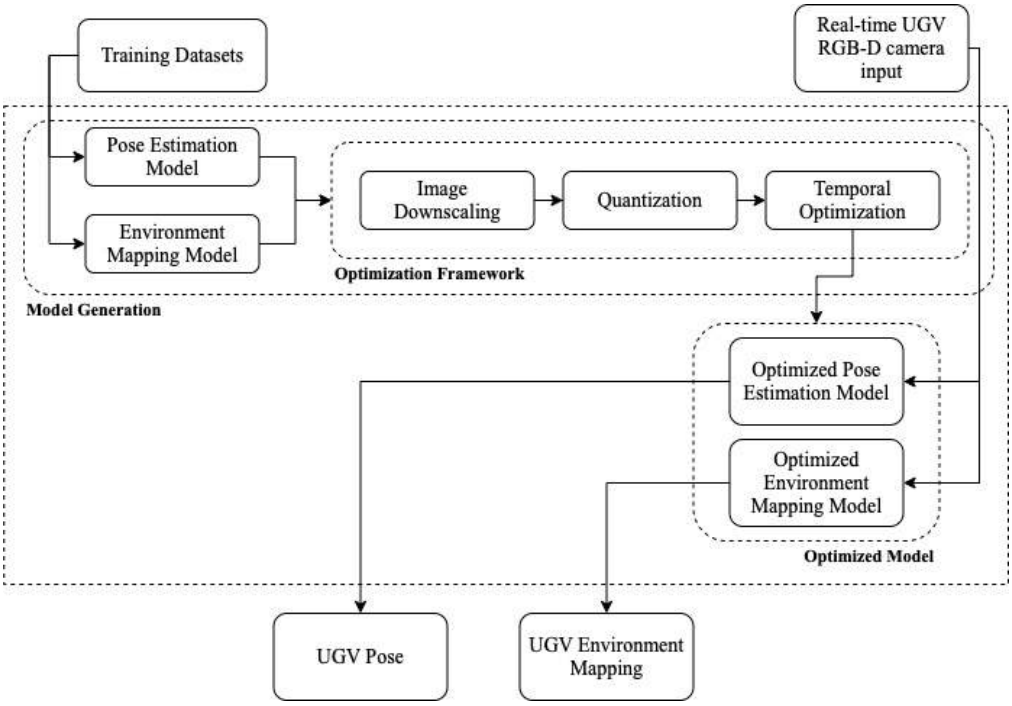


Figure 9: Final Model Design

Although a loss of accuracy is inevitable when dealing with optimization for resource-constrained hardware, the feature of having a dictionary for various budget constraints allows us to mitigate that issue. In conditions that are easy for the model to evaluate and achieve high accuracy in, the UGV robot can utilize more energy efficient models by using a stricter budget. However, the option to utilize a more powerful model is still available in the case when it is needed by relaxing the budget constraint and trading off some energy instead.

9. Ethics

9.1 Safety of SLAM in Public Areas

Due to being limited by the available datasets and computational resources required to develop deep learning models for very large amounts of video, our DL-SLAM model will not be always able to generalize to every encounter in the real world. This touches on the ethical aspect of deploying fully autonomous systems without sufficient testing due to limitations pertaining to the technology we currently have for testing. To solve this issue, we will be using a separate risk assessment module that overrides the actions of the DL-SLAM model. This will use classical techniques of object detection and segmentation in computer vision to detect people and assess any risk of harm. Thus, the UGV robot will behave in a more safe and robust way.

9.2 Safety of SLAM in Bad Environmental Conditions

Our DL-SLAM model will have degraded performance in bad environmental conditions such as fog or rain. This, as in 9.1, will also cause a public hazard as the UGV robot might not behave as expected and pose a risk to public property and people around it. To help mitigate this problem, we will also train the model on noisy data with unclear and blurry landmarks to adapt the model to such environmental conditions. Since we also have a dictionary for optimized sub-models given different budget constraints, we can add a module that selectively chooses a more powerful model (albeit less energy efficient) in order to combat the loss of accuracy in its measurements and behave in a more robust and expected manner.

9.3 Privacy Issues with the Use of RGB-D Cameras in Public

The deployment of autonomous robots with advanced RGB-D cameras in public might pose an issue for privacy depending on the region. This ethical aspect will vary greatly from one region or country to another and the context where the model is deployed. However, in areas where this poses a privacy issue, the party deploying our model has to comply with the ethical and legal standards in the region. From our side, we have to ensure that the UGV robot is secure, such that attackers cannot access potentially private information. To do this, we will make sure to deploy our model with the most recent version of ROS with minimum vulnerabilities and keep pushing frequent updates to keep up with the security fixes.

Additionally, it is important to develop clear and transparent standards and guidelines for the use of machine learning algorithms. These standards and guidelines should specify the ethical principles and values that should be taken into account, and should provide guidance on how to design and evaluate machine learning algorithms to ensure that they are safe and reliable. This is especially important in the current climate where there is little to no oversight over the use of such technologies and heavily vary from location to another as discussed earlier.

10. Implementation

10.1 Description of Implementation

- *Optimization of DROID SLAM:*

During our optimization process for the Droid SLAM model, we explored various techniques to improve performance. We first attempted to optimize the framework by downscaling the images captured. The images at the beginning were being processed at 640x480 pixels. We scaled them

down to 480x360 pixels and the model was still able to generate an accurate 3d map of the environment as well as perform accurate pose estimation throughout the map.

We also used quantization to reduce the memory footprint of the model by running the neural network with 16 and 8 bit integers rather than the default 32 bit floating point values. The memory used decreased by 1GB with the 16 bit optimization and around 1GB again with 8 bit optimization. The model was also still able to generate accurate 3D maps with these optimizations.

Since the model has an internal cache for the depth estimation and image stream as well as the 3D point cloud of the environment, reducing the rate of the image stream with temporal optimization can help further reduce the memory used. Although the aim was to achieve a 30fps latency, we tried to go down to 20fps, which lowered the memory used by around 1GB.

We also accurately calibrated the sensors to align measurements with the actual physical environment. To do that, we took measurements of the coefficients for the camera model used. Those measurements can help us to preprocess the image stream to the SLAM model, and improve our results. The most important values are horizontal and vertical focal lengths, fx and fy, and vertical and horizontal principal points, cx and cy. In addition, distortion intrinsic to the lens used in the camera impacts the captured images. Specifically, we needed the radial distortion coefficients, k1 and k2, as well as the tangential distortion coefficients, p1 and p2. We used a checkerboard pattern printed on an a4 paper and took a video of it from multiple angles. Then we ran the calibration code using OpenCV, which measures the distances between the corners to obtain the camera lens intrinsic values.

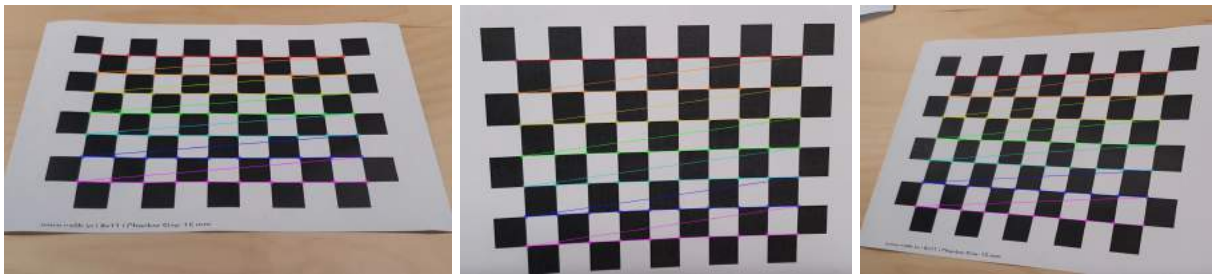


Figure 10: Sample images from calibration run on the camera

For 640x480 and 480x360 resolutions, we obtained the following results:

Camera Intrinsic Value	640x480	480x360
fx	680.02484259	490.85720450
fy	680.02484259	490.85720450
cx	327.68913016	244.64718560
cy	247.97807184	180.72367538
k1	$2.34372360 \times 10^{-1}$	2.195733×10^{-1}
k2	$-3.1994138 \times 10^{-2}$	-2.573023×10^{-2}
p1	$3.36162435 \times 10^{-3}$	4.682945×10^{-3}
p2	$7.38297722 \times 10^{-3}$	5.2389259×10^{-3}

Table 4 : Camera intrinsic values for each resolution

- *Hardware Implementation:*

The next stage of the implementation was running the new optimized algorithm on an older version of Ubuntu (18.04). This is due to the fact that Husky's most documented simulation exists in ROS1 Melodic and ROS1 Melodic is compatible only with Ubuntu 18.04. Moreover, this was also done to sync our system specifications with that running on the lab Husky robot. The NYUAD researchers already had experimentation done on Husky so this route would have made debugging and troubleshooting easier in case of running into errors.

Once the installation of the new Operating System was complete, we attempted to implement the optimized algorithm. However, we quickly ran into issues with missing NVIDIA drivers. We found that the latest drivers on which the optimized algorithm runs were not compatible with the older version of Ubuntu. We then attempted to bridge the issue by downgrading some of the dependencies to their earlier versions and reducing the minimum required RAM. The issue still remained unresolved as the downgrading was unsuccessful. This proved to be a major bottleneck as testing progress for the optimized model was halted.

In an attempt to circumvent the issue, an attempt was made to upgrade the system on Husky instead to Ubuntu 20.04 to sync up with the workspace system. This route also fell through in the end due to network connection issues. The robot needed connection to the NYU Wifi for ROS

communication with the optimized-model-running-workspace which could not be provided due to technical complications. Therefore, a physical implementation of the optimized model on the Husky robot was not achieved and testing was conducted through other alternative avenues described in Section 10.3.

10.2 Issues Faced and Changes Made During Implementation:

For the optimization framework, NetAdapt, we ran into several errors while first running the code. The versions of the dependencies used were outdated now since the github repository is over 2 years old, and we had to change the dependencies and modify the code to get it running. For the dependencies, we changed the following:
- Pillow downgraded to version <7
- PyTorch upgraded to 1.12.1
- Torchvision upgraded to 0.13.1
- Cudatoolkit upgraded to 11.6

For the code errors, we had to convert all instances of [kept_filter_idx] and [kept_filter_idx_fc] in the functions.py file to [kept_filter_idx].long() and [kept_filter_idx_fc].long(). With those changes, the NetAdapt algorithm was functional.

We also experienced issues with the operating system kernel, which was not compatible with the optimized algorithm. To address this, we referred to the open-source code and README file provided with the algorithm and made necessary changes to the system requirements. We disabled certain features and attempted to run the algorithm with bare-minimum system features to avoid complications with unrelated programs, but we still encountered issues with the algorithm crashing during runtime. The algorithm crash would also disable the OS GUI, an issue that was only resolved by resetting the progress and reinstalling the OS.

We referred to the provided open-source code and README file for guidance. Despite making the recommended changes to dependencies and installing the prescribed NVIDIA drivers on the new startups, we had to reset and reinstall the system 3 times after which we gave the system to a senior researcher at our lab who took it upon himself to manually troubleshoot the system with correct dependencies. Despite our best efforts to deploy the optimized algorithm, we were unable to successfully run it on the older version of Ubuntu. This phase proved to be a bottleneck issue and it took up massive time and effort without allowing for any progress to be made.

10.3 Initial Results:

We ran the model on our own data with videos taken around campus as seen in the figures below.

Figure 11: Sample images from indoors environment going in a circle (A5 building)

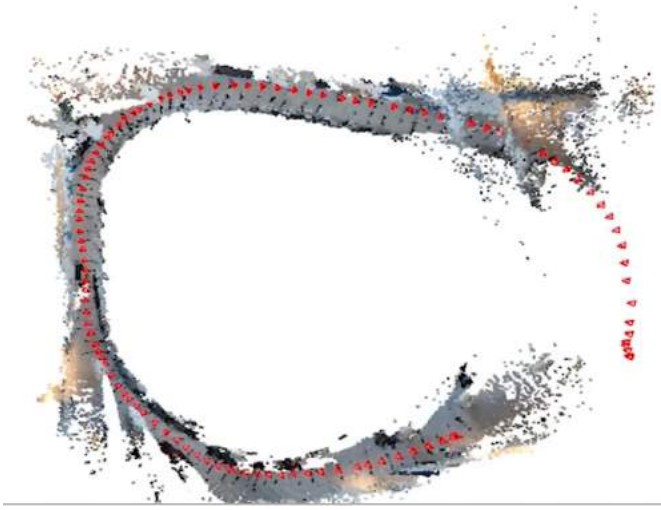


Figure 12: Generated 3D map with poses (red triangles) of the A5 building.

Figure 13: Sample images from NYUAD campus outdoors.

Figure 14: Generated 3D map with poses (red triangles) of a section of NYUAD campus outdoors

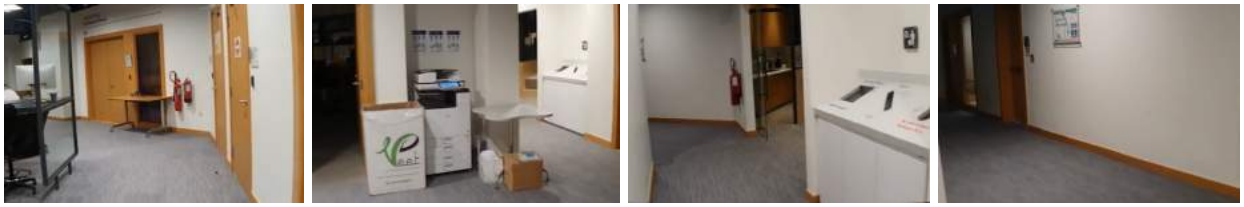


Figure 15: Sample images from the cybersecurity lab.

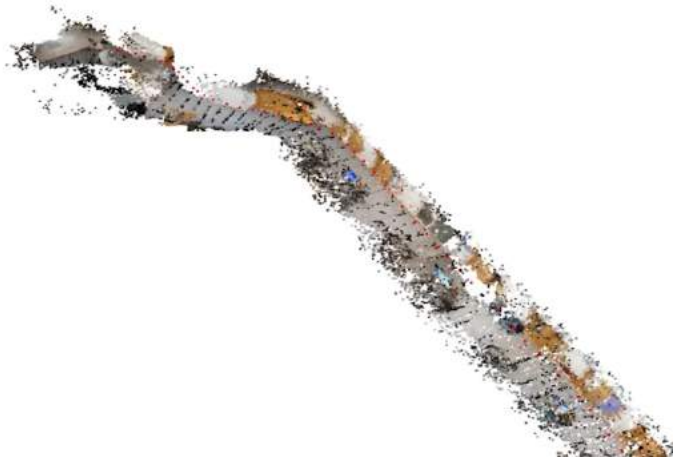


Figure 16: Generated 3D map with poses (red triangles) of the cybersecurity lab corridor

11. Design Evaluation

11.1 Criteria for Testing

11.1.1 Accuracy

The model will be tested against many datasets such as the RGB-D dataset to assess the accuracy of our model. The accuracy for pose estimation will be evaluated using the Absolute Trajectory Error (ATE) in centimeters as described in [9]. The expected baseline accuracy for pose estimation is 5cm.

11.1.2 Latency

Latency will be measured in frames per second (fps). The expected latency that the model should be designed for is 33ms (or 30fps) in order to fully utilize the depth information from the RGB-D camera.

11.1.3 Power Usage

This will be measured in watts using active power consumption of the GPU while running the model. The expected power consumption is 10W while the DL-SLAM model is running on Jetson TX2 GPU.

11.2 Test Data

Using 8-bit quantization, the impact on ATE is not very significant for most environments. In some cases it achieves a slightly better or worse performance in very small margins. Therefore, we utilized 8-bit quantization as there was no significant accuracy loss while also providing us with a lower memory footprint.

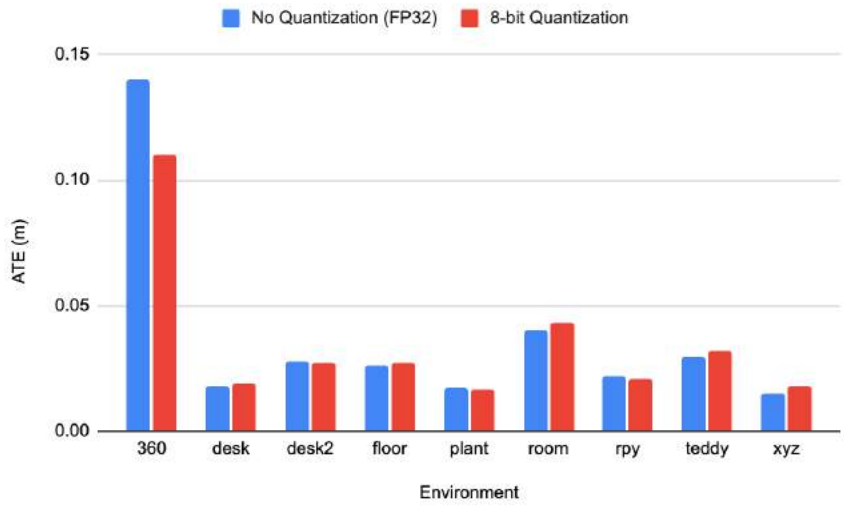


Figure 17: Absolute trajectory error (ATE) in meters for different environments using the default FP32 model representation and 8-bit quantization

The decrease in resolution, on the other hand, did produce some increase in the error, but it was also still a small increase aside from the first outlier in the 360 environment. This also made it favorable for us to use 480x360 resolution as we lowered the memory usage again for a small increase in the error. Additionally, the lower resolution would also decrease the latency as the model is processing 44% less pixels.

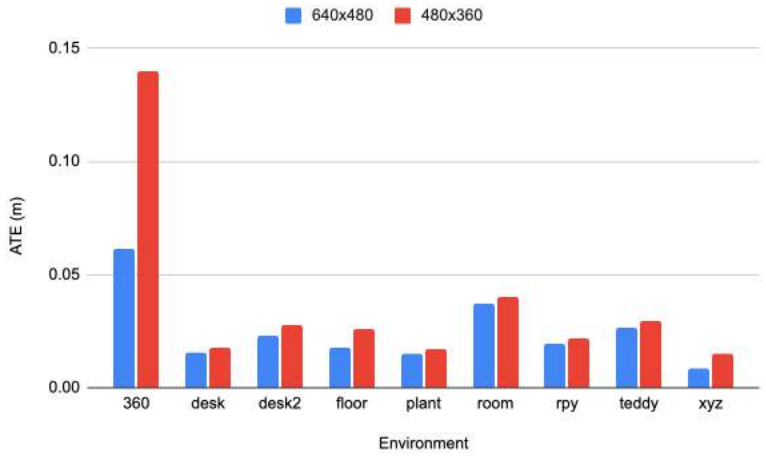


Figure 18: ATE in meters for different environments using 640x480 and 480x360 resolution

Figures 19 and 20 below show the impact of adjusting the buffer size used in the model to hold the images being utilized together on the VRAM usage as well as the ATE. The impact of the buffer size on the accuracy seems minimal since figure 20 shows that the error is almost constant regardless of the buffer size. On the other hand, the impact of the VRAM usage is much more pronounced. Since we observed no error increase, we decided to go with the 256 buffer size, which brought down our VRAM usage to 6,058 MB only. We tried going even lower, but the model was giving us several errors about lacking enough elements to process in some of the environments.

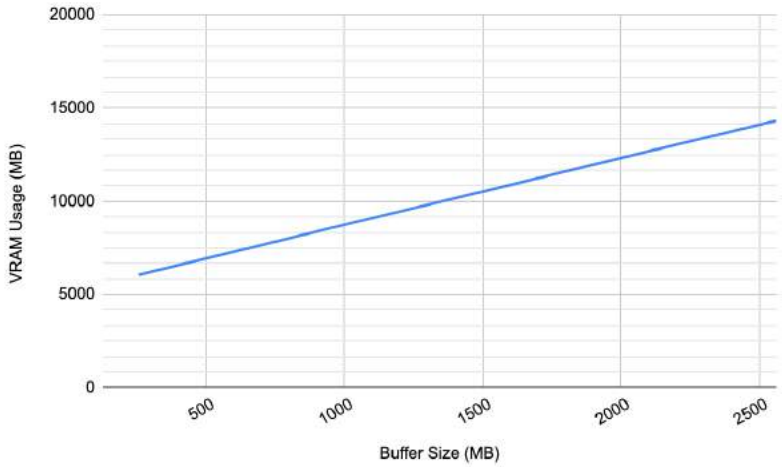


Figure 19: GPU VRAM usage in MB of the model with different buffer sizes

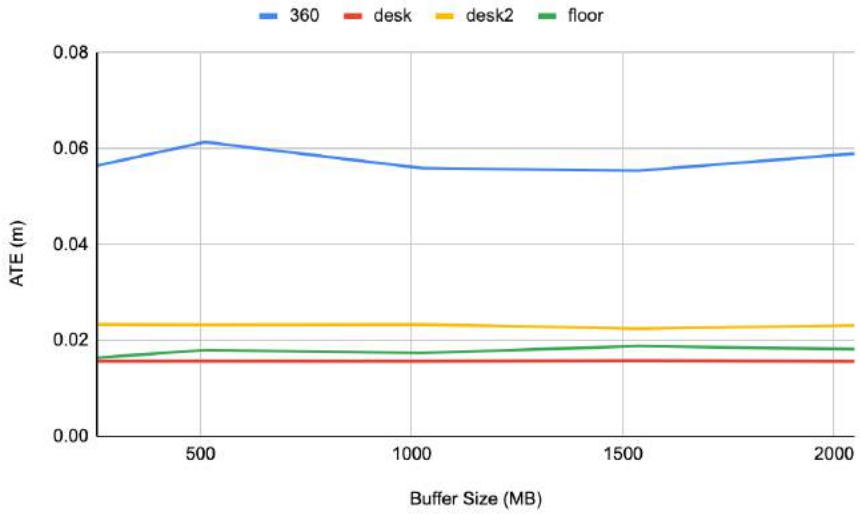


Figure 20: ATE in meters for increasing buffer size in MB in different environments

11.3 Discussion of Test Data

For the accuracy, we have met the testing criterion of 5cm apart from one of the environments tested (360 environment in the RGB-D dataset). However, the custom tests we had with our videos around campus had a higher error than those in the public datasets used. For instance, the pose estimation as seen in figure x in the indoors circle movement, the map does not fully connect at the end and the pose drifts by around 30-40cm. The model generated better results with less quick turns and was best with smooth movement in a semi-straight line (less than 60 degree rotations and with slow smooth rotations), such as the cybersecurity corridor test and the campus outdoors.

We suspect that the reason for this is that the camera calibration is not completely accurate. In fact, running the calibration on different videos using the same camera generates different values for the camera intrinsics with around 5-10% difference per run for the focal lengths (fx and fy) and the principal points (cx and cy). When we tested those different camera intrinsic values, we also obtained slightly different 3d maps and pose estimations. Therefore, the reason for the worse results when we run our videos could be the inaccurate calibration of the camera. For the future, we could perform a more robust calibration by using a significantly larger number of images of 1000+ since we only used 100-150 frames for our calibration.

Another interesting area to research for such models, would be adding radial and tangential distortions as well as distorting the image by focal length in the datasets used for training. This might help the model be more resistant to small changes in the calibration values and generate more robust results.

This issue of quick rotations also appears in simulated environments like Unity. It requires further investigation because the unity simulations have perfect calibration since the same camera properties are used digitally. We have attempted to resolve this problem by increasing the frame rate of the camera and even running the model offline, not in real time, but the issue was still present.

Regarding the latency criterion, the model runs on an RTX 3090 with around 50fps. However, for the Jetson TX2 GPU, we lowered it to 20fps and also used a window of 2 images, which instead samples an average of 2 images at once rather than processing each image individually. These adjustments would make it possible for the model to run on the Jetson TX2 reasonably well at 20fps. While we did not satisfy the 30fps/33ms requirement, the results with 20fps and two image averages still produced good results for the 3d map and pose estimation.

As we have successfully brought down the memory footprint down to 6GB, making the model capable of being run on the Jetson TX2 GPU smoothly, and consequently on Clearpath husky, the power consumption criterion was met as the TDP of the Jetson TX2 GPU is 10W. With all the initial testing criteria and design constraints satisfied, we were able to successfully design an optimized DL-VSLAM algorithm that is robust yet computationally-cheap and energy-efficient at the same time.

12. Bill of materials

In terms of software frameworks or libraries, only open-source software and datasets are used for this project. For hardware, all of the items listed below are already available on campus and nothing needs to be purchased. However, the following will be used in this project for reference.

#	Item name	URL	Price (USD)
1	Jetson TX2 Module	https://www.digikey.com/en/products/detail/seeed-technology-co.,-ltd/102110402/16570938	$650.00
2	Husky UGV	https://www.generationrobots.com/en/402177-husky-a200-ugv-mobile-base.html	$26767.19
3	Intel RealSense Depth Camera D435	https://www.amazon.com/Intel-Realsense-D435-Webcam-FPS/dp/B07BLS5477?source=ps-sl-shoppingads-lpcontext&ref_=fplfs&psc=1&smid=A29CFOW4UM3L3L	$310.19

Works Cited

1. Wheelock, Clint. “Computer Vision Hardware, Software, and Services Market to Reach $26.2 Billion by 2025, According to Tractica.” *Businesswire*, 6 Mar. 2018,
2. Alzubaidi, Laith, et al. "Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions." Journal of big Data 8.1 (2021): 1-74.
3. Chaplot, Devendra Singh, et al. "Learning to explore using active neural slam." *arXiv preprint arXiv:2004.05155* (2020).
4. Yang, Tien-Ju, et al. "Netadapt: Platform-aware neural network adaptation for mobile applications." *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.
5. Gordon, Ariel, et al. "Morphnet: Fast & simple resource-constrained structure learning of deep networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.
6. “Datasets - RGB-D SLAM Dataset and Benchmark.” *Technical University of Munich*, 8 Mar. 2016, https://vision.in.tum.de/data/datasets/rgbd-dataset.
7. “SUN3D Database.” *Princeton University*, The Trustees of Princeton University, https://sun3d.cs.princeton.edu/.
8. “NYU Depth Dataset V2.” *NYU*, https://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html.
9. Sturm, Jürgen, et al. "A benchmark for the evaluation of RGB-D SLAM systems." *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2012.
10. Teed, Zachary, and Jia Deng. "Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras." *Advances in Neural Information Processing Systems* 34 (2021): 16558-16569.
11. Czarnowski, Jan, et al. "Deepfactors: Real-time probabilistic dense monocular slam." *IEEE Robotics and Automation Letters* 5.2 (2020): 721-728.
12. Teed, Z., & Deng, J. (2021). Droid-slam: Deep visual slam for monocular, stereo, and rgb-d cameras. Advances in neural information processing systems, 34, 16558-16569.

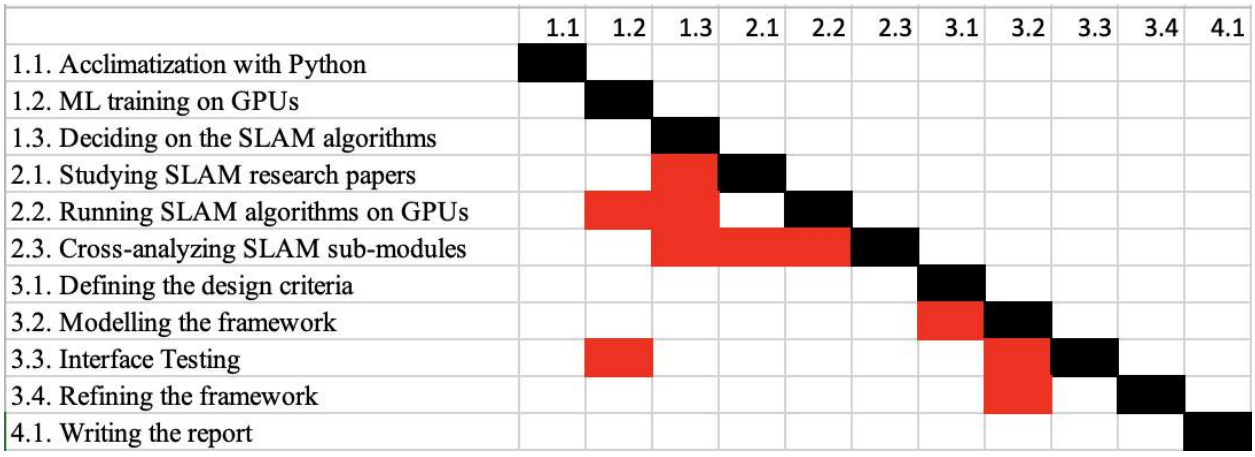
Appendix A. Project Management

A.1 Work Breakdown Structure

Primary Task	Duration (days)	Start	End
1. Background and Literature Review	21	8/22/2022	9/12/2022
1.1. Acclimatization with Python	7	8/22/2022	8/29/2022
1.2. ML training on GPUs	7	8/29/2022	9/5/2022
1.3. Deciding on the SLAM algorithms	7	9/5/2022	9/12/2022
2. Comparative Analysis of algorithms	42	9/12/2022	10/24/2022
2.1. Studying SLAM research papers	14	9/12/2022	9/26/2022
2.2. Running SLAM algorithms on GPUs	14	9/26/2022	10/10/2022
2.3. Cross-analyzing SLAM sub-modules	14	10/10/2022	10/24/2022
3. Design of Optimization Framework	42	10/24/2022	12/5/2022
3.1. Defining the design criteria	7	10/24/2022	10/31/2022
3.2. Modelling the framework	7	10/31/2022	11/7/2022
3.3. Interface Testing	14	11/7/2022	11/21/2022
3.4. Refining the framework	14	11/21/2022	12/5/2022
4. Capstone Report	7	12/5/2022	12/12/2022
4.1. Writing the report	7	12/5/2022	12/12/2022

Work breakdown structure for spring semester

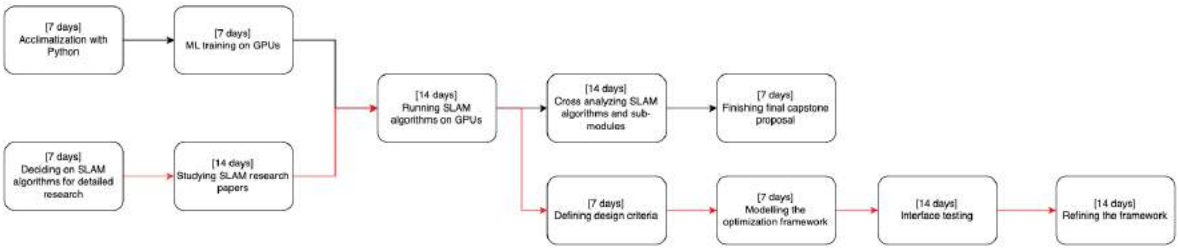
A.2 Design Structure Matrix



Design structure matrix for spring semester

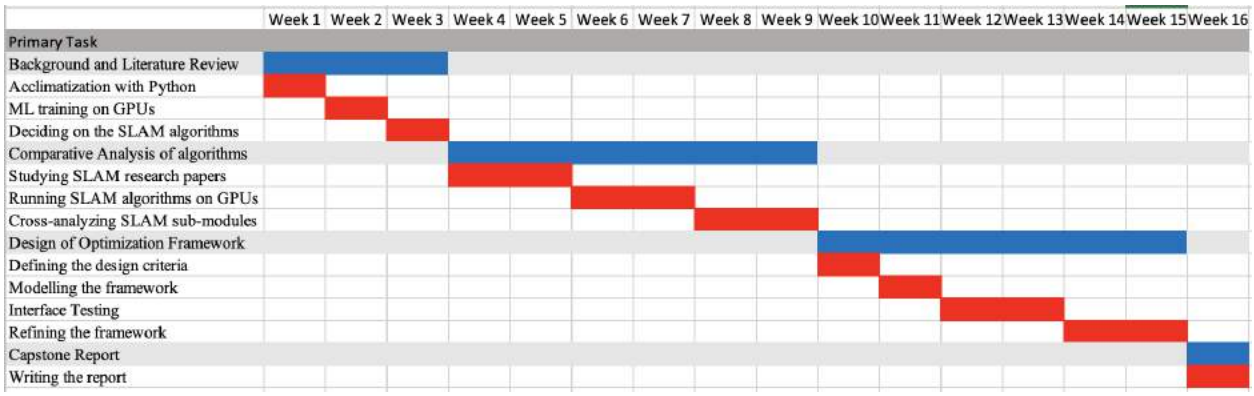
A.3 Critical Path

The critical path is highlighted in red, totalling 77 days.



Critical path figure for spring semester

A.4 Gantt Chart



Gantt chart for spring semester