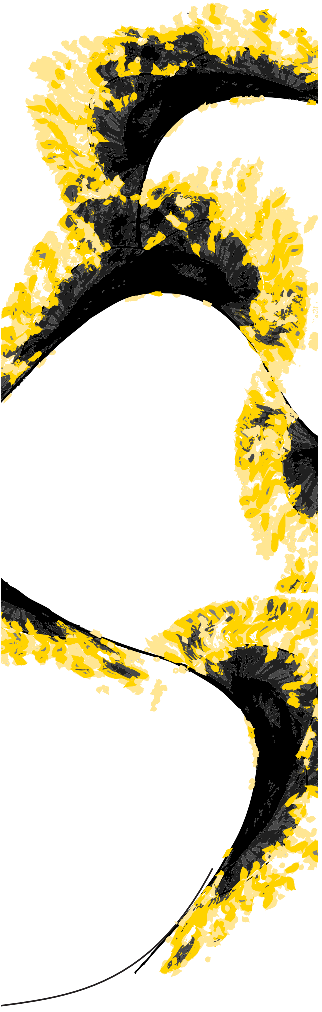


ROBOTICS
AND
MECHATRONICS

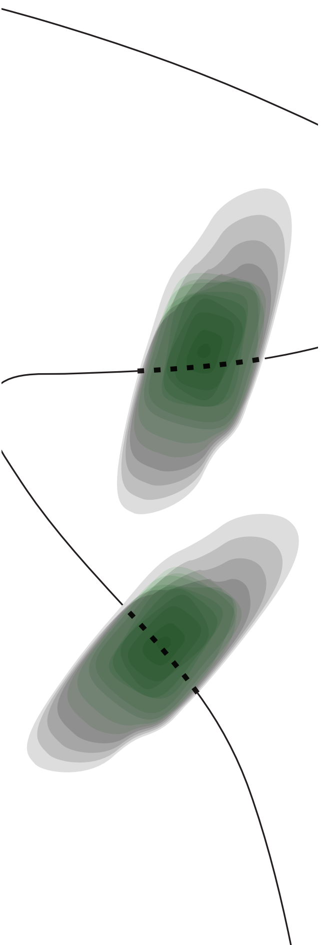
ACCELERATED IMPLEMENTATION OF 3D VISUAL EKF SLAM FOR HANDHELD PERFUSION IMAGING

S.H.G. (Stijn) Wolters

MSC ASSIGNMENT

March, 2021

015RaM2021
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands



UNIVERSITY OF TWENTE. | TECHMED CENTRE
UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

This page is intentionally left blank.

Abstract

This thesis presents a fast implementation of the Visual Error State Extended Kalman Filter Simultaneous Localization and Mapping (ES-EKF SLAM) algorithm for the handheld perfusion imaging device (HAPI). The ES-EKF SLAM algorithm can be used to increase the accuracy of the handheld perfusion imaging, while providing a 3D map of the scanned surface. Unfortunately, the ES-EKF SLAM algorithm has a high order computational complexity. The computational bottlenecks of the algorithm are mapped through a worst-case computational time complexity analysis and a runtime analysis of an existing software implementation. Through an analysis of the functionality of the ES-EKF SLAM implementation for the HAPI, a set of system requirements is composed. Through a design space exploration, the GPU with Tensor cores and CUDA cores is then selected to accelerate the parts of the ES-EKF SLAM algorithm with the highest time complexity. The computation actions are allocated on a system consisting of two computers: a single board computer with a GPU that performs the landmark measurement acquisition and a desktop computer with a high-end GPU that performs the ES-EKF SLAM computations. Through runtime evaluation of the accelerated implementation and an existing non-accelerated implementation, it can be concluded that the GPU accelerated implementation has a higher runtime performance than a non-GPU accelerated implementation. However, in order to satisfy the runtime requirements of the system, the input size must be limited. In the future implementations opportunities for further acceleration of the algorithm must be taken in consideration.

Table of Contents

1 Introduction

In the dermatology field, procedures such as lesion measurement and tracking, procedure planning and real-time assistance, and clinical and patient education can benefit from the introduction of enhanced reality (Sharma, Vleugels, & Nambudiri, 2019). The Handheld Perfusion Imaging device (HAPI) is a device under development that aims at non intrusively measuring and tracking skin lesions. It does this by using laser speckle contrast imaging and 3D mapping to project the measured perfusion map onto the measured surface. An issue in the techniques of handheld perfusion imaging is the introduction of additional motion artifacts caused by the operator of the device and the movement of the patient. Such movement artifacts compromise the functional performance of the perfusion measurement and are needed to be compensated for (Chizari, Knop, Sirmacek, van der Heijden, & Steenbergen, 2020). Therefore, non-intrusive measurement of the movement artifacts is essential.

Through visual means of sensing, such as the optical flow for sensing translational velocity, this non-intrusive measurement can be performed. As described in Appendix 1, optical flow is currently being implemented on the HAPI. However, another method of visual movement measurement that additionally generates a 3D map of the measured surface exists. This method is called is Visual Simultaneous Localization and Mapping (SLAM). As the name suggests, SLAM performs simultaneous pose estimation and mapping of landmarks in 3-dimensional space. A system for this has been proposed by Shah (2020) and Grimm (2020) which uses Error State Extended Kalman Filter SLAM (ES-EKF SLAM). Unfortunately, this algorithm is computationally expensive and executing the algorithm for densely mapping a detailed surface in real-time is challenging, as the current implementation is too slow for practicality. Therefore, the goal of this thesis is to accelerate this ES-EKF SLAM algorithm using computer hardware. A number of implementations have already aimed at improving the runtime performance of EKF slam which will be outlined in the next section.

1.1 Related Work

Multiple accelerations of existing SLAM algorithms exist based on different methods; one of which is decreasing the computational complexity of the SLAM algorithm. The EKF-SLAM algorithm is based on Bayesian filtering, while other algorithms rely on different filtering methods such as the FastSLAM and SEIF SLAM algorithms. FastSLAM, for instance, uses particle filtering to represent probability distributions by using a finite set of sample states instead of the probability density defined by a mean and covariance of the Extended Kalman Filter (Montemerlo & Thrun, 2007). SEIF SLAM uses a sparse information filter to represent the state probability using a sparse information matrix (Thrun, et al., 2004). This project focusses on accelerating the execution of the Visual ES-EKF SLAM algorithm, so work related to the acceleration of this algorithm is focused on in this section. Other attempts at implementations of fast extended Kalman filter-based SLAM algorithms can be classified as using algorithm optimizations or using specialized hardware to accelerate the algorithms using custom computer architectures.

Tertei, Piat & Devy (2016) designed a hardware accelerator for the EKF SLAM algorithm which accelerates the matrix multiplication using a systolic array on a Zynq-7020 FPGA. They divided the SLAM algorithm into two parallel processes: the front-end, where the computer vision algorithms for landmark detection are performed, and the back end, where the filtering is performed. These parts are executed in pipeline, in order to increase the throughput. This implementation takes in account the symmetry of the cross-covariance matrix in order to reduce computational time and memory usage. The implementation achieves a throughput of 44 frames per second with a map size of 70 landmarks, where only 20 landmarks are being updated per timestep. If all landmarks are being updated per timestep, their real-time requirement of 30 frames per second is still being met for a map size of 60 landmarks.

More recently, de Souza Rosa, Dasu, Doniz & Bonata (2018) use a Faddeev Systolic Array implemented on a FPGA to compute Schur's complement in order to accelerate the EKF *update* step and analyze the use of fixed-point and single precision floating point number representation on their space- and time efficiency. The implementation used pipelined arithmetic units in order to achieve a higher computational frequency and compared the use of manufacturer-supplied division processing elements against manually implemented division processing elements. They conclude that the floating-point representation achieves a higher computational frequency and uses less resources on the FPGA than the fixed-point representation. Unfortunately, no conclusions are drawn about the runtime performance of the implementation for EKF-SLAM computations.

Idris, Arof, Noor, Tamil, & Razak (2012) increase the performance of a monocular EKF-SLAM algorithm by optimizing the matching algorithm for landmark association. Their implementation uses the characteristic that features can be processed independently in order to parallelize the algorithm. The resulting architecture consists of two FPGA execution engines which each process 10 features simultaneously. The performance of this implementation is 25 times faster than the software implementation.

Piat, Fillatreau, Tortei, Brenot, & Devy (2020) later designed an implementation which accelerates the feature detection, matching and *update* algorithms of the EKF-SLAM on a FPGA. During this process, a co-design method was proposed which was then used to iteratively implement and evaluate the different hardware accelerators. During the first iteration, a dataflow model was used in order to expose parallelism to decrease latency within the system. The resulting model holds for any filtering technique and feature detection and landmark matching methods. During the next iteration, a hardware accelerator for the FAST feature detection algorithm and tessellation algorithm which divides the image into grid cells was implemented. The latter of which is used to decrease the image processing complexity as the image processing algorithms are only applied to a subset of the image and to select a single feature from each grid cell. During the third iteration, a hardware accelerator for the *update* step within the EKF algorithm was implemented. During the fourth iteration, a hardware accelerator for the feature matching was implemented based on the matching algorithm for the BRIEF feature descriptor. The final result containing all optimizations is able to process 24 frames per second where 20 landmarks are updated during each SLAM iteration.

Another accelerated implementation of an accelerated SLAM algorithm is described by Li, et al. (2020). Instead of an FPGA, the GPU (a computer architecture which allows for highly parallel computations) is used. In this case, the Multicol-SLAM algorithm is optimized by introducing multithreaded pipelining in order to optimize computational efficiency by utilizing both the GPU and CPU in the system and by implementing GPU acceleration of the ORB feature detection, extraction, and matching algorithms. Additionally, the characteristics of the fisheye camera lens are used to decrease the image size and therefore decrease the computational complexity of the image processing algorithms, while acquiring images from multiple cameras. The implementation is based on the NVIDIA Jetson Tegra X1, which is an embedded single board computer and is able to process on average 15 frames per second.

Giubilato, Chiodini, Pertile & Debei (2019) analyze different stereo visual SLAM and visual odometry algorithms which are supported by ROS (Robotics Operating System) with GPU optimizations on an embedded platform, the NVIDIA Jetson Tegra X2. The SLAM algorithms include ORB-SLAM2, S-PTAM, LibVISO2, RTAB-MAP, and ZED-VO. The implemented optimization is to perform image processing algorithms such as image undistortion, stereo rectification, and contrast normalization. This optimization reduces the load on the CPU, which increases the processed frames per second. They also include that performing contrast normalization and image rectification increases the estimation accuracy and robustness to lighting conditions.

Paz, Piniés, Tardós & Neira (2008) implement a fast EKF-SLAM algorithm on a general-purpose CPU by extending the algorithm with a divide and conquer algorithm. This algorithm keeps the state vector low by dividing the global map into local maps which are smaller and conditionally independent. The maps can be joined together using a hierarchical tree structure based on common landmarks in the maps. The experimental results showed that while the time needed to perform a SLAM iteration was occasionally high, the researchers expect that a multithreaded and precompiled solution could reach real-time execution speeds with updates at 25 Hz.

The above implementations show promising acceleration of various SLAM algorithms by exploiting various techniques, mainly including parallelization, both on data- and thread-level using parallel execution units. However, most discussed implementations only accelerate the front-end computer vision algorithms, without considering the acceleration of the filtering algorithm. Except the implementations proposed by de Souza Rosa et. al. (2018) and Paz et al. (2008), which accelerate the EKF-SLAM algorithm using algorithmic acceleration, while the prior also utilizes specialized hardware. Furthermore, only Piat et. al. provide a fully accelerated implementation of the EKF-SLAM algorithm where both the front-end and back-end are accelerated. Their resulting implementation only allows for a very sparse map of the environment with only 20 landmarks.

1.2 The contents of this report

The goal of this thesis is to develop an accelerated implementation of the proposed Visual ES-EKF SLAM algorithm for the HAPI. The result is a GPU-based system, which utilizes data-level parallelism in the ES-EKF SLAM algorithm and the computer vision algorithms for landmark finding. The ES-EKF SLAM computations are performed on a desktop computer with a GPU and the landmark measurement acquisition is performed on a single board computer with an on-board GPU. Both operations are executed in parallel in order to maximize the computer utilization. The system is able to execute 10 SLAM iterations per second, while tracking 1800 landmarks.

The project follows a structured process using the following guidelines:

- Analyze and determine the computational bottlenecks of ES-EKF SLAM.
- Determine the requirements of the ES-EKF SLAM implementation.
- Select applicable optimization methods.
- Determine the computing hardware.
- Allocate the ES-EKF SLAM computations.
- Design the ES-EKF SLAM system for HAPI.
- Implement the designed ES-EKF SLAM system.
- Evaluate the accelerated system with regards to runtime performance and fulfillment of the determined requirements.

These guidelines are further described and follows in this report. The structure of this report is as follows:

Chapter 2 provides background information on the ES-EKF SLAM algorithm, a comparison between the performance of matrix-matrix multiplications (which are the most common operations in the ES-EKF SLAM algorithm) on the GPU versus the FPGA and provides applicable optimization methods for accelerating the ES-EKF SLAM algorithm using the GPU.

Chapter 3 provides the problem analysis. During this analysis, the computational bottlenecks of the ES-EKF SLAM algorithm are mapped. The application of the ES-EKF SLAM algorithm for the HAPI is then analyzed in order to determine which requirements the ES-EKF SLAM implementation must meet to be functional for the HAPI. The chapter results by determining the requirements of the ES-EKF SLAM implementation.

Chapter 4 provides the design space exploration, during which the computing hardware for the ES-EKF SLAM implementation is selected and the computations are allocated. Determination of the computing hardware is first done by selecting an applicable computer architecture, deciding between a computer implemented on an FPGA, or the GPU which provides CUDA cores and Tensor cores. Then the allocation of the computing action is performed, resulting in a rough system design. This system design is then used to select the computing hardware containing the selected computer architecture for optimal execution of the ES EKF-SLAM algorithm.

Chapter 5 proposes the resulting system design based on the results of the design space exploration and the system requirements. The chapter then describes the implementation method and the resulting implementation.

Chapter 6 evaluates the runtime performance of the system compared to an existing MATLAB implementation of the ES-EKF SLAM algorithm and a non-GPU accelerated implementation. This chapter also provides a worst-case execution time analysis. This analysis is used to determine the maximum size of the input of the landmark measurement acquisition implementation and the maximum number of tracked landmarks by the ES-EKF SLAM implementation in order to meet the execution rate requirements set in Chapter 3.

Chapter 7 provides a discussion of the runtime evaluation and the worst-case execution time analysis and ties the findings to the algorithm complexity analysis performed in Chapter 3.

Chapter 8 concludes the report by summarizing the resulting system implementation and the met requirements. This is followed by a discussion of future recommendations.

2 Background

This chapter provides background information on the ES-EKF SLAM algorithm, the performance of GPUs compared to FPGAs for matrix-matrix multiplications and applicable optimization methods that can be used during the design and implementation of an accelerated ES-EKF SLAM implementation.

2.1 The SLAM Algorithm

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in robotics which arises when a robot does not know its pose and has no access to a map of the environment, as described by Thrun, Burgard, & Fox (2006). A solution to the SLAM problem uses measurements from, for example, accelerometers, gyroscopes, LIDAR, or camera systems to derive its pose and make a map of its environment. The SLAM problem does not just exist in robotics, but also in terrain mapping or augmented reality applications, like the application of SLAM in the HAPI device.

This section describes the Error State Extended Kalman Filter Visual SLAM algorithm discussed by Van der Heijden (2018) and Shah (2020).

2.1.1 The goal of SLAM

The result of SLAM is a state vector $x(t)$ containing the pose of the device $y(t)$ at time t and a map of K landmarks m_1 ... m_K. These landmarks are defined as points in 3D Euclidean space. The pose of the device is defined as a position in 3D space expressed in world coordinates $p^w(t)$, quaternion rotation expressed in world coordinates $q^w(t)$ and translational and angular velocities of the camera relative to world coordinates, but expressed in camera coordinates, $v_{wc}^c(t)$ and $\omega_{wc}^c(t)$ respectively. The state vector looks as follows:

$$x(t) = \begin{bmatrix} y(t) \\ m_1 \\ \vdots \\ m_k \end{bmatrix} \tag{1}$$

With

$$y(t) = \begin{bmatrix} p^w(t) \\ q^w(t) \\ v_{wc}^c(t) \\ \omega_{wc}^c(t) \end{bmatrix} \tag{2}$$

The vector $y(t)$ is 13-dimensional. As the position $p^w(t)$ of the device is defined as a point in 3D Euclidian space, its vector is also 3-dimensional. The linear velocity, $v_{wc}^c(t)$ is therefore also defined as a 3D vector. The rotation, however, is described as a quaternion and is therefore 4-dimensional with unit length. This representation is the most robust way of representing orientation in 3D space. The angular velocity $\omega_{wc}^c(t)$ on the other hand is described as a 3D vector.

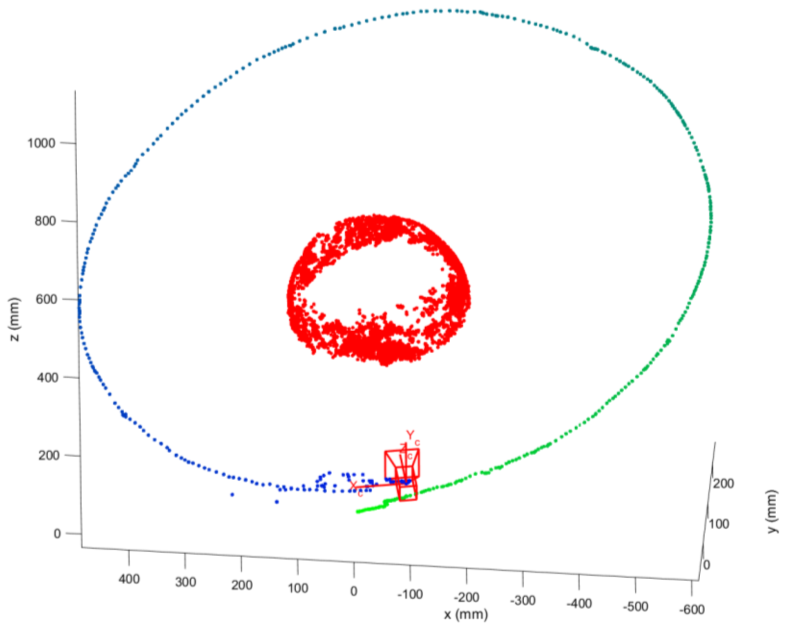


Figure 2.1: An example of the result of ES-EKF SLAM: a mapped surface of a globe (red) with the camera path (green/blue) and final pose (red).

Figure 2.1 shows an example of the result of the ES-EKF SLAM algorithm where the camera rotates around a globe. The red points in the middle represent the mapped landmarks m_1 ... m_K and are surface points of the globe. The points depicted in the green/blue hue depict the camera position over time $p^w(t)$. The red camera shows the last measured pose of the camera $p^w(t)$ and $q^w(t)$.

Because measurement uncertainty due to noise propagates, the state vector contains uncertainty as well. Therefore, the state is expressed as a mean, the state vector, and a covariance, covariance matrix which both describe a probability distribution. Here the mean is used as the most likely pose and map. The covariance matrix is 12+3K-dimensional and is defined as follows:

$$C = \begin{bmatrix} C_p & 0 & & & & \cdots & 0 \\ 0 & C_q & 0 & & & \cdots & 0 \\ & 0 & C_v & 0 & & \cdots & 0 \\ & & 0 & C_\omega & 0 & \cdots & 0 \\ & & & 0 & C_{m_1} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & C_{m_k} \end{bmatrix} \tag{3}$$

The resulting probability distribution of the state vector is based on measurements $z_{1:K}(1{:}t)$, a measurement for each landmark at different points in time. Therefore, defining the following probability distribution function:

$$p(x(t)|z_{1:K}(1{:}t))$$

This probability distribution function gives the likelihood of different poses and landmark positions given the measurements. The higher the probability, the more likely the corresponding pose is given the measurements. Therefore, the result of SLAM is the state vector with the highest probability:

$$y, m_{1:K} = armax\left(p\left(x(t)\middle|z_{1:K}(1:t)\right)\right) \tag{4}$$

As stated before, it suffices to describe the gaussian probability density with a mean and a covariance. The mean of $p\big(x(t)|z_{1:K}(1:t)\big)$ will therefore be denoted as $\hat{x}(t|t)$, which is the mean of the state vector at time t given the measurements at time t. The covariance matrix of the state vector will have the same notation: $C(t|t)$, which is the covariance matrix of the state vector at time t given the measurements at time t.

2.1.2 The flow of EKF SLAM

The EKF SLAM algorithm is iterative. This means that every time the algorithm is executed, previous results are combined with new data to compute the current state. The flow of EKF SLAM consists mainly of a cycle during which new measurements are taken, the state vector is updated using these measurements, and the next state vector is predicted. The last two steps are characteristic for the Kalman Filter that is used. However, the initialization step is also important as this step defines the starting pose of the device and its uncertainty parameters.

The cycle of the algorithm is shown in Figure 2.2.

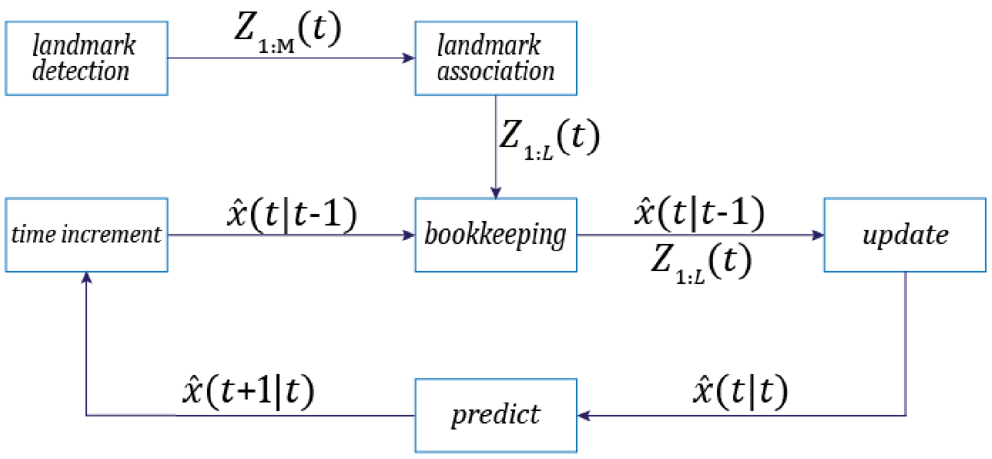


Figure 2.2: The dataflow of EKF-SLAM.

2.1.2.1 Initialization

The initialization of the algorithm defines the state at the very first cycle at $t = 1$. Here the pose of the robot is defined as follows: $p^w(1) = [0 \quad 0 \quad 0]^T$, $q^w(1) = [1 \quad 0 \quad 0 \quad 0]^T$, $v_{wc}^c(1) = [0 \quad 0 \quad 0]^T$, and $\omega_{wc}^c(1) = [0 \quad 0 \quad 0]^T$. As no landmarks have been observed yet, the list of landmarks is empty. Therefore, the state vector is equal to the pose and velocity of the device: $x(1) = y(1)$. This is the state vector at time $t = 1$.

The initial covariance matrix also contains just the covariances of the pose and velocity of the device. Because the initial pose of the device defines the world coordinate system where the future poses and landmark positions are represented in, the covariance parameters for the pose are set to 0. The covariance matrices for the velocities C_v and $C_\omega$depend on the uncertainty of movement during the initialization; the higher the uncertainty that there was no movement, the lower the covariances. The initial covariance matrix is then defined as follows:

$$C(1) = \begin{bmatrix} 0_{3x3} & 0 & \cdots & 0 \\ 0 & 0_{4x4} & 0 & 0 \\ \vdots & 0 & C_v & \vdots \\ 0 & 0 & \cdots & C_\omega \end{bmatrix} \tag{5}$$

2.1.2.2 Measurement acquisition and Bookkeeping

After initialization, the iterative SLAM cycle is entered. The first step in this cycle is the acquisition of measurements. This will be discussed further in Section 2.1.3. The resulting measurements $z_{1:M}(t)$ go through a *bookkeeping* algorithm where the measurements are associated with the landmarks in the pool and landmark duplicates are detected and removed. The *bookkeeping* algorithm then removes landmarks from the state vector if necessary and adds new landmarks to the state vector. The measurements $z_{1:L}(t)$ and previous state vector $\hat{x}(t|t-1)$ and its covariance matrix $C(t|t-1)$ are then used to update the state.

2.1.2.3 State Update

During the *update* step of the Kalman Filter, the latest measurements are used to update the state vector. This means that the current state vector, which is a prediction of the pose of the device based on previous measurements, $\hat{x}(t|t-1)$ and $C(t|t-1)$, are updated to take the current measurements $z_{1:L}(t)$ in account, resulting in $\hat{x}(t|t)$ and $C(t|t)$ which is the mean state vector at the current time given the measurements at the current timestep.

The state vector that results from the *prediction* step contains the pose of the device and the map of the environment at that timestep. This state vector can therefore be used for its desired application.

2.1.2.4 State Prediction

When the state vector at the current timestep is computed given the latest measurements: $\hat{x}(t|t)$ and $C(t|t)$, it is time to predict the next pose of the device: $\hat{x}(t+1|t)$ and $C(t+1|t)$. This prediction uses the motion model of the device. This includes predicting the change of pose and the velocity of the camera system, without updating the map. The resulting state vector is then used in the next iteration as the state vector at that time step based on all previous measurements: $\hat{x}(t|t-1)$ and $C(t|t-1)$.

2.1.3 Acquiring landmark measurements

Camera measurements are done to obtain visual landmarks. This is not part of the ES-EKF SLAM algorithm, but it is worth to be discussed, as it is part of the process to derive the state vector from visual measurements. Taking these measurements makes use of a single camera (mono) or multiple cameras (stereo). As the algorithm that is being discussed in this report was developed to use a stereo camera setup, that is the method that will be discussed here. From this stereo camera setup, two images can be acquired simultaneously. From these two images, 3D landmarks in the form of points in the 3D space relative to the device can be derived.

2.1.3.1 Calibrated cameras; before measurement

Before visual measurement acquisition can be started, it is important to have the cameras calibrated so that their intrinsic parameters and their extrinsic parameters are known. The intrinsic parameters of a camera describe the parameters of a pinhole camera, which transforms a point in the 3D Euclidean world to a projection to a 2D Euclidean plane, as shown in Figure 2.3.

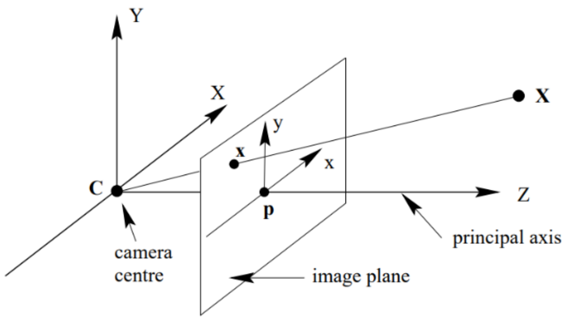


Figure 2.3: Diagram of the pinhole camera model (Hartley & Zisserman, 2004).

The intrinsic parameters are denoted by the matrix K (Hartley & Zisserman, 2004):

$$K = \begin{bmatrix} \alpha_x & S & x_0 \\ 0 & \alpha_y & y_o \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Here, the parameters α_x and α_y denote the focal length of the camera in pixels in the horizontal and vertical direction respectively; $\alpha_x = fm_x$ and $\alpha_y = fm_y$, where f is the focal length and $m_{x,y}$ is the number of pixels per unit length. The number of pixels per unit length might differ per direction, depending on the shape of the pixels. Therefore, the number of pixels per unit length is separated in the x direction and the y direction.

x_o and y_0 denote the horizontal and vertical position of the principal point with respect to the image plane in pixels; $x_0 = p_x m_x$ and $y_0 = p_y m_y$. Here, p_x and p_y are the horizontal and vertical position of the principal point in unit length.

Finally, the parameter S in the calibration matrix denotes the skew of the pixels, which is nonzero if the pixels are not perpendicular in the x- and y-axis. This can often be neglected as this almost never occurs in camera systems.

The extrinsic parameters contain information about the position of the center of the camera with relation to some coordinate system. Like the pose of the camera device as described before, the pose of a camera consists of a translation t^w and a rotation R^w expressed in world coordinates. t^w is a 3D vector, denotes the location of the camera and R^w is a rotation matrix, denoting the rotation of the camera relevant to the world coordinate system. In stereo camera systems, one camera coordinate system is often used as the world coordinate system so that the other camera pose is expressed in this coordinate system. The extrinsic parameters of the camera that is used as a reference are therefore set to 0.

After calibrating both cameras, their intrinsic and extrinsic parameters are known. The intrinsic parameters are denoted by K_1 and K_2. The extrinsic parameters are denoted by R_1^w and R_2^w for their rotations and by t_1^w and t_2^w for their translations.

2.1.3.2 Keypoint detection

Now that the camera parameters are known, landmark measurement can be done. This is done by first finding landmarks on each image through a keypoint detection algorithm, resulting in two sets of keypoints. The keypoints from each set are then matched with each other, so that keypoint pairs are formed of keypoints which are visible on each image. These keypoints can then be turned into the 3D landmarks that are used as measurements.

2.1.3.3 From 2D keypoint to 3D landmark

Localizing the keypoints in 3D space is done through Minimum Mean Square Error (MMSE) estimation, which assumes that there is a linear relationship between the measurement and the unknown parameter (van der Heijden, 3D measurements from camera images, 2016). This relationship describes how a physical point is projected on an image plane:

$$z = HX^w + n \tag{7}$$

Here, z is the observed measurement, the location of the landmark on the image plane in pixels. X^w is the unknown parameter vector, the location of the landmark in the 3D Euclidean world coordinates. H is the measurement matrix, which describes the theoretical projection of the 3D point on the image plane. n denotes measurement noise, which is added to the signal, which is assumed to be gaussian with zero mean and a covariance matrix C_n. The covariance of the noise C_n depends on the variance in the uncertainty of the pixel localization in all directions σ^2 and the a priori depth estimation Z:

$$C_n = Z\sigma^2 I \tag{8}$$

The algorithm for MMSE also assumes that prior knowledge about the position of the point is available. This prior knowledge is defined by a gaussian probability distribution, which is defined by a mean position $\hat{X}^w_{prior}$ and a covariance matrix C_{prior}.

First, a single camera is used to estimate the location of the landmark. This is done by first defining the relation between the 3D landmark and the 2D keypoint:

$$p^1 = K_1(R_1^w X^w + t_1^w) \tag{9}$$

This Equation can be rewritten to look more like Equation 7 by rewriting K:

$$K = \begin{bmatrix} k_1^T \\ k_2^T \\ k_3^T \end{bmatrix} \tag{10}$$

Where k_n^T is the nth row of matrix K.

This can be used to rewrite Equation 7 by also substituting p^1:

$$\begin{bmatrix} \alpha p_x^1 \\ \alpha p_y^1 \\ \alpha \end{bmatrix} = \begin{bmatrix} k_1^T \\ k_2^T \\ k_3^T \end{bmatrix} (R_1^w X^w + t_1^w) \tag{11}$$

Where p_x^1 and p_y^1 are the x and y coordinates in the image frame in pixels of the keypoint.

Without going into the mathematical proof, Equation 9 can be rewritten in the form of Equation 7:

$$z_1 = H_1 X^w + n_1 \tag{12}$$

With:

$$z_1 = \begin{bmatrix} k_1^T - \hat{p}_x^1 k_3^T \\ k_2^T - \hat{p}_y^1 k_3^T \end{bmatrix} t_1^w \tag{13}$$

$$H_1 = \begin{bmatrix} \hat{p}_x^1 k_3^T - k_1^T \\ \hat{p}_y^1 k_3^T - k_2^T \end{bmatrix} R_1^w \tag{14}$$

$$n_1 = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \end{bmatrix} Z^1 \tag{15}$$

Where ε_1 and ε_2 are the localization errors in the horizontal and vertical directions.

Now, to make an estimate of the location of the landmark the following algorithm is used:

Operation	Equation
Compute the innovation matrix.	$S = HC_{prior}H^T + C_n$ (16)
Compute the Kalman Gain matrix.	$G = C_{prior}H^T S^{-1}$ (17)
Estimate the landmark position using prior knowledge.	$\hat{X}^w_{post} = \hat{X}^w_{prior} + G\left(z - H\hat{X}^w_{prior}\right)$ (18)
Update the landmark position uncertainty.	$C_{post} = C_{prior} - GSG^T$ (19)

Algorithm 2.1: Algorithm for estimating the 3D position of a point based on 2D pixel coordinates.

Algorithm 2.1 results in a first posterior estimate of the location of the landmark expressed in world coordinates as a gaussian probability distribution defined by a mean $\hat{X}^w_{post}$ and a covariance C_{post}. This is a rough estimate, as the prior knowledge is not accurate and only includes a line from the principal point of the camera to the landmark and excludes a good estimate of the depth. A second estimate can be performed using the second camera to make a cross section of the area where the landmark might be to make a better estimate of its location, and mainly its depth.

Making a better estimate using the second camera is done by another iteration of Algorithm 2.1, but using the posterior location and covariance $\hat{X}^w_{post}$ and C_{post} from the first iteration as prior knowledge $\hat{X}^w_{prior}$ and C_{prior} for the second iteration.

For the second camera, the linear relationship stated in Equation 7 is as follows:

$$z_2 = H_2 X^w + 2 \tag{20}$$

With:

$$z_2 = \begin{bmatrix} k_1^T - \hat{p}_x^2 k_3^T \\ k_2^T - \hat{p}_y^2 k_3^T \end{bmatrix} t_2^w \tag{21}$$

$$H_2 = \begin{bmatrix} \hat{p}_x^2 k_3^T - k_1^T \\ \hat{p}_y^2 k_3^T - k_2^T \end{bmatrix} R_2^w \tag{22}$$

$$n_2 = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \end{bmatrix} Z^2 \tag{23}$$

Where k_n^T are derived from the intrinsic parameter matrix K_2.

This results in an estimate for the location of the landmark $\hat{X}^w_{prior}$ expressed in world coordinates and the uncertainty of this measurement C_{prior}, which can be used for the EKF SLAM Algorithm. Doing this for all keypoints results in a list of measurements $z_{1:K}(t)$ containing the estimated location of the landmark and its uncertainty $C_{1:K}$. This list contains all measurements for all landmarks at time t, which might include measurements of landmarks that are not visible at this time. These measurements are then empty. New observed landmarks are added to the list of measurements and old ones can be deleted to lower the computational cost.

2.1.4 The Error State Extended Kalman Filter

The Visual SLAM algorithm makes use of the Error State Extended Kalman Filter, ES-EKF, for its state estimation. Like the Extended Kalman Filter, the ES-EKF performs its state estimation in two steps: *prediction* and *updating*. However, the ES-EKF splits the estimated state of the system up in two parts: the estimated nominal state and the estimated error state as follows:

$$\hat{x}(t|t) = \breve{x}(t|t) + \hat{\tilde{x}}(t|t) \tag{24}$$

Where $\hat{x}(t|t)$ is the estimated state, $\breve{x}(t|t)$ is the nominal state, and $\hat{\tilde{x}}(t|t)$ is the estimated error. This defines the error in the system Equation, which will be discussed later in this chapter. The error state is what the Kalman filter is applied to, with the resulting error estimation being used to compensate the nominal state to estimate the real state of the system.

2.1.5 Updating

The *update* step of the Kalman filter uses the visible landmarks to provide a better estimate of the current state $\hat{x}(t|t)$ based on the visible measurements $z_{1:L}(t)$ and the previous prediction of the state $\hat{x}(t|t-1)$. The measurements $z_{1:L}(t)$ are the L visible measurements within $z_{1:K}(t)$. This process is based on a linear approximation using a truncated Taylor series expansion evaluated at the most recent estimate of the state:

$$\begin{aligned} z(t) &= h\big(x(t)\big) + v(t) \\ &= h(\hat{x}(t|t-1) + \varepsilon) + v(t) \\ &\approx h\big(\hat{x}(t|t-1)\big) + H\varepsilon + v(t) \end{aligned} \tag{25}$$

where $h\big(x(t)\big)$ estimates the position of the landmark based on the pose of the device. H is the Jacobian matrix of $h\big(x(t)\big)$ evaluated at $\breve{x}(t|t-1)$. ε is the predicted error of the state and $H\varepsilon$ represents the uncertainty in the predicted state. $v(t)$ represents measurement noise, which is zero-mean and has a covariance C_v.

Using this model, a predicted measurement $h\big(\breve{x}(t)\big)$ can be computed based on the current state and the visible landmarks. This predicted measurement can be compared to the actual measurement $z(t)$ to compute the error state measurement $\tilde{z}(t)$, which is then used to compute the error state $\hat{\tilde{x}}(t|t)$, which is the difference between the previously estimated state and the current state and can therefore be used to compute the updated state.

First, an estimate of the measurement based on the landmarks and the current pose is computed with $h\big(x(t)\big)$ to update the state.

$$\begin{aligned} h_k\big(x(t)\big) &= \begin{bmatrix} -R_w^T & 0_{3\times(7+3k)} & R_w^T & 0_{3\times(K-k)} \end{bmatrix} \begin{bmatrix} p^w(t) \\ q^w(t) \\ v_{wc}^c(t) \\ \omega_{wc}^c(t) \\ m_1 \\ \vdots \\ m_k \\ \vdots \\ m_K \end{bmatrix} \\ &= R_w^T\big(m_k - p^w(t)\big) \end{aligned} \tag{26}$$

This Equation expresses the landmarks relative to the camera coordinate system using the following conversion between the quaternion and the rotation matrix:

$$R_w = \begin{bmatrix} 1-2({q_2}^2+{q_3}^2) & 2(q_0q_3+q_1q_2) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2({q_1}^2+{q_3}^2) & 2(q_0q_1+q_2q_3) \\ 2(q_0q_2+q_1q_3) & 2(q_2q_3+q_0q_1) & 1-2({q_1}^2+{q_2}^2) \end{bmatrix} \tag{27}$$

This results in a vector of size $3L$ which is then used to compute the error state measurement vector:

$$\tilde{z}(t) = z(t) - h\big(\breve{x}(t)\big) \tag{28}$$

The error state vector is then multiplied by the Kalman Gain matrix, which is calculated as follows:

First, calculate the Jacobian of the measurement function:

$$H = \frac{\partial}{\partial x^T} h(x)\bigg|_{\breve{x}(t|t-1)} \tag{29}$$

Which is given by:

$$H_k\big(x(t)\big) = \begin{bmatrix} -R_w^T & 2\big[h_k\big(x(t)\big)\big]_\times & 0_{3\times(3-3k)} & R_w^T & 0_{3\times(K-k)} \end{bmatrix} \tag{30}$$

This results in a $3K \times 3L$ matrix, which is then used to compute the innovation matrix S:

$$S = HC(t|t-1)H^T + C_v \tag{31}$$

Which of size $3L \times 3L$ and where C_V is the $3L \times 3L$ covariance matrix containing the covariances of the measurement noise as follows:

$$C_V = \begin{bmatrix} C_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & C_L \end{bmatrix} \tag{32}$$

Then the Kalman Gain matrix is computed based on the previously predicted uncertainty and the computed innovation:

$$G = C(t|t-1)H^T S^{-1} \tag{33}$$

The Kalman Gain matrix is of size $3K + 12 \times 3L$ is then used to update the estimated error state, which is then used to correct the nominal state vector:

$$\hat{\tilde{x}}(t|t) = G\tilde{z}(t) \tag{34}$$

$$\breve{x}(t|t) = \breve{x}(t|t-1) + \hat{\tilde{x}}(t|t) \tag{35}$$

Finally, the uncertainty is updated using the innovation matrix and the Kalman Gain matrix.

$$C(t|t) = C(t|t-1) - GSG^T \tag{36}$$

The resulting nominal state, $\breve{x}(t|t)$ and $C(t|t)$, is the most recent estimation and can be considered to be the current state of the system. This is therefore the output of the algorithm which can be used for its intended application.

2.1.6 Prediction

The *prediction* phase of the Kalman Filter predicts what the next state of the system will be based on the kinematics of the device. This results in a prediction of the pose and velocity of the system during the next time step:

$$p^w(t+1) = p^w(t) + R_w v_{wc}^c(t)\Delta \tag{37}$$

$$q^w(t+1) = \exp(W\Delta)q^w(t) \tag{38}$$

$$v_{wc}^c(t+1) = \Gamma v_{wc}^c(t) + w_1(t) \tag{39}$$

$$\omega_{wc}^c(t+1) = \Lambda\omega_{wc}^c(t) + w_2(t) \tag{40}$$

Where $R_w$is given by Equation 27 and W is given by:

$$W = \frac{1}{2}\begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \\ -\omega_x & 0 & \omega_z & -\omega_y \\ -\omega_y & -\omega_z & 0 & \omega_x \\ -\omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{41}$$

$w(t)$ is the process noise, which explains the uncertainty within the system Equation and results from inaccurate position and velocity measurement. The process noise is assumed to be normally distributed, has a 0-mean and a covariance matrix C_w given by:

$$C_w = \begin{bmatrix} 0_{7\times7} & 0_{7\times6} & & \\ 0_{6\times7} & C_1 & 0_{3\times3} & 0_{13\times3K} \\ & 0_{3\times3} & C_2 & \\ & 0_{3K\times13} & & 0_{3K\times3K} \end{bmatrix} \tag{42}$$

Where C_1 and C_2 represent the diagonal covariance matrices which describe the uncertainty of the velocity.

The Equations above can be transformed into the following linear function:

$$x(t+1) = f\big(x(t)\big) + w(t) \tag{43}$$

Where $f\big(x(t)\big)$ is the system Equation, which describes the kinematical model:

$$f\big(\check{x}(t|t)\big) = \begin{bmatrix} I_{3\times3} & 0_{3\times4} & R_w\Delta & 0_{3\times3} & 0_{3\times3K} \\ 0_{4\times3} & \exp(W\Delta) & 0_{4\times3} & 0_{4\times3} & 0_{4\times3K} \\ 0_{3\times3} & 0_{3\times4} & \Gamma & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times4} & 0_{3\times3} & \Lambda & 0_{3\times3K} \\ 0_{3K\times3} & 0_{3K\times4} & 0_{3K\times3} & 0_{3K\times3} & I_{3K\times3K} \end{bmatrix} \begin{bmatrix} p^w(t) \\ q^w(t) \\ v^c_{wc}(t) \\ \omega^c_{wc}(t) \\ m_1 \\ \vdots \\ m_K \end{bmatrix} \tag{44}$$

This Equation is used during the *prediction* step to predict the next nominal state as follows:

$$\check{x}(t+1|t) = f\big(\check{x}(t|t)\big) \tag{45}$$

Finally, the uncertainty of the next state needs to be changed accordingly. This is done using the Jacobian matrix of $f(x(t))$ evaluated at $\hat{x}(t|t)$:

$$F = \frac{\partial}{\partial x^T} f(x)\bigg|_{\hat{x}(t|t)} \tag{46}$$

This Jacobian matrix is given by:

$$F\big(\hat{x}(t|t)\big) = \begin{bmatrix} I_{3\times3} & P_\theta & R_w\Delta & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & I_{3\times3} & 0_{3\times3} & \frac{1}{2}I_{3\times3}\Delta & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times3} & \Gamma & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & \Lambda & 0_{3\times3K} \\ 0_{3K\times3} & 0_{3K\times3} & 0_{3K\times3} & 0_{3K\times3} & I_{3K\times3K} \end{bmatrix} \tag{47}$$

With:

$$P_\theta = -2\Delta R_w[v^c_{wc}(t)]_\times \tag{48}$$

Which after computation can be used to compute the uncertainty of the predicted state, together with the uncertainty of the process noise C_w:

$$C(t+1|t) = FC(t|t)F^T + C_w \tag{49}$$

Now the next state is predicted and can be updated using a new set of measurements during the next timestep.

2.1.7 Summary of ES-EKF SLAM computations

To summarize, Algorithm 2.2 and Algorithm 2.3 provide an overview of the *update* and *predict* steps of the Error State Extended Kalman Filter.

Operation	Equation	
Compute $h(x(t))$	$h_k(x(t)) = R_w^T(m_k - p^w(t))$	(1)
Compute the Jacobean $H(x(t))$	$H_k(x(t)) = \begin{bmatrix} -R_w^T & 2[h_k(x(t))]_\times & 0_{3\times(3-3k)} & R_w^T & 0_{3\times(K-k)} \end{bmatrix}$	(2)
Compute the measurement covariance matrix	$C_V = \begin{bmatrix} C_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & C_L \end{bmatrix}$	(3)
Compute measurement error	$\tilde{z}(t) = z(t) - h(\check{x}(t))$	(4)
Compute the innovation	$S = HC(t\|t-1)H^T + C_v$	(5)
Compute the Kalman Gain	$G = C(t\|t-1)H^T S^{-1}$	(6)
Compute the estimated error state	$\hat{\tilde{x}}(t\|t) = G\tilde{z}(t)$	(7)
Update the nominal state with the estimated error state	$\check{x}(t\|t) = \check{x}(t\|t-1) + \hat{\tilde{x}}(t\|t)$	(8)
Update the covariance matrix	$C(t\|t) = C(t\|t-1) - GSG^T$	(9)

Algorithm 2.2: Overview of the ES-EKF Update algorithm

Operation	Equation
Compute W	$W = \frac{1}{2}\begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \\ -\omega_x & 0 & \omega_z & -\omega_y \\ -\omega_y & -\omega_z & 0 & \omega_x \\ -\omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \qquad (10)$
Compute $f(\breve{x}(t \mid t))$	$f(\breve{x}(t \mid t)) = \begin{bmatrix} I_{3\times3} & 0_{3\times4} & R_w\Delta & 0_{3\times3} & 0_{3\times3K} \\ 0_{4\times3} & \exp(W\Delta) & 0_{4\times3} & 0_{4\times3} & 0_{4\times3K} \\ 0_{3\times3} & 0_{3\times4} & \Gamma & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times4} & 0_{3\times3} & \Lambda & 0_{3\times3K} \\ 0_{3K\times3} & 0_{3K\times4} & 0_{3K\times3} & 0_{3K\times3} & I_{3K\times3K} \end{bmatrix} \begin{bmatrix} p^w(t) \\ q^w(t) \\ v_{wc}^c(t) \\ \omega_{wc}^c(t) \\ m_1 \\ \vdots \\ m_K \end{bmatrix} \qquad (11)$
Predict the nominal state for the next time step	$\breve{x}(t+1 \mid t) = f(\breve{x}(t \mid t)) \qquad (12)$
Compute P_θ	$P_\theta = -2\Delta R_w[v_{wc}^c(t)]_\times \qquad (13)$
Compute the Jacobean $F(\hat{x}(t \mid t))$	$F(\hat{x}(t \mid t)) = \begin{bmatrix} I_{3\times3} & P_\theta & R_w\Delta & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & I_{3\times3} & 0_{3\times3} & \frac{1}{2}I_{3\times3}\Delta & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times3} & \Gamma & 0_{3\times3} & 0_{3\times3K} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & \Lambda & 0_{3\times3K} \\ 0_{3K\times3} & 0_{3K\times3} & 0_{3K\times3} & 0_{3K\times3} & I_{3K\times3K} \end{bmatrix} \qquad (14)$
Predict the covariance matrix of the next time step	$C(t+1 \mid t) = FC(t \mid t)F^T + C_w \qquad (15)$

Algorithm 2.3: Overview of the ES-EKF Prediction algorithm

2.2 The performance of BLAS on GPU versus FPGA

This section provides a comparison of the performance of basic linear algebra subroutines (BLAS) for GPU implemented computer architectures discussed in Appendix 2 and FPGA implemented computer architectures based on existing literature. As explained later in this report, the most computational expensive parts of the ES-EKF SLAM algorithm are data independent and mainly consist of the matrix-matrix operations. It is therefore of importance to understand this performance difference for different computer architectures.

A literature review has been performed by Escobar, Chang, & Valderrama (2016) which analyzed high productivity computing (HPC) platforms and algorithms and concluded that dense linear algebra operations are computed most efficiently in GPUs. This is supported by Jones, Powell, Bouganis & Cheung (2010) who have compared the productivity of an NVIDIA GPU using CUDA cores and a multiple-FPGA computer and concluded that the GPU has a significantly higher performance on large dense matrix-matrix multiplications and scalar-sum computation of a vector. More recently, Minhas, Bayliss & Constantinides (2014) compared the performance of matrix-matrix multiplication implementations of standard and non-standard precision computations on an NVIDIA GPU using CUDA cores and an FPGA. They have concluded that for larger matrices, their GPU implementation outperforms the FPGA platform for all investigated precision computations.

Escobar et al. (2016) also concluded that sparse linear algebra operations are performed most efficient in FPGAs. This is supported by Nurvitadhi et al. (2017) who have evaluated machine learning algorithms on two generations of Intel FPGAs against a state-of-the-art NVIDIA GPU. During this research, the performance of matrix-matrix multiplication algorithms was evaluated for dense and sparse matrices and with different data types. They concluded that while GPUs outperform FPGAs for dense matrix-matrix multiplications, FPGAs outperform GPUs for sparse matrix-matrix multiplications.

Newer generations of NVIDIA GPUs introduce the Tensor core, a new type of processing unit which performs matrix-multiply-and-accumulate on 4x4 matrices per clock cycle. These type of processing units increase the performance for matrix multiplications when compared to CUDA cores as concluded by Mukunoki, Ozaki, Ogita & Imamura (2020). They present a method to use the lower precision Tensor core for FP64 matrix-matrix multiplication and conclude that even though the arithmetic units in Tensor cores compute FP16 inputs with FP32 accuracy, the Tensor cores can still be used to operate on FP64 number representations with a higher performance than CUDA cores.

2.3 Applicable optimization methods

This section discusses different optimization methods that can be applied when developing an accelerated implementation of the ES-EKF SLAM algorithm. The optimization methods regard optimizing software for the CUDA compatible GPU and global optimization methods that can apply to the whole system in order for them to be used in the system design and implementation.

2.3.1 Optimize for the most common case

First of all, as recommended by Hennessy & Patterson (2012), it is important to focus on the common case while designing an optimized system. This means that for every development iteration, it is important to implement an acceleration for the computations that are performed the most and take longest to compute in order to gain the highest performance increase.

2.3.2 Decrease problem size

Decreasing the problem size is defined as decreasing the size of the input of the different parts of the system. This includes the image size for the landmark measurement acquisition, maximum number of landmark measurements for both the data transmission and the landmark association algorithm, and the maximum number of visible landmarks l and number of landmarks in the pool k. Especially the latter will decrease and limit the computational complexity of the ES-EKF algorithm.

2.3.3 Maximize Utilization

Maximizing utilization is defined as utilizing all processing devices in the system as much as possible, so that all computation resources are used at all times. This can be done by utilizing thread-level parallelism by executing independent computations on different processors. As described by the NVIDIA Corporation (2021), each processor should be used for the type of work it is optimal for, such as sequential algorithms being assigned to the host processor, and parallel algorithms being assigned to the device and utilizing data-level parallelism by performing parallelized algorithms on independent data. Also, thread level parallelization can be utilized in order to perform the execution of unrelated algorithms simultaneously if not all computational resources are used.

2.3.4 Minimize data transmission

Data transmission can potentially take a large amount of time to finish, especially for large data structures such as the state covariance matrix, or for slow transmission rates over Wi-Fi. In order to minimize data transmission latency, the amount of data being transmitted is to be minimized. As discussed above, this can be done by limiting the number of landmark measurements such that not too many measurements are being sent over the network to the desktop computer. Also, the host processor and GPU transmit data for the computation of the ES-EKF SLAM. As the state covariance matrix is desired to be as large as possible, it is important to prevent communication of this matrix between device and host. Therefore, the matrix can be initialized in GPU memory and all operations on the matrix are performed by the GPU.

2.3.5 Maximize instruction throughput

As discussed by the NVIDIA Corporation (2021), maximizing instruction throughput entails minimizing the instructions with low throughput, minimizing warp divergence, and reducing the number of instructions. Minimizing the instructions with low throughput means to trade precision for speed, which could increase the runtime performance, sacrificing the accuracy of the qualitative result of the SLAM algorithm. Minimizing warp divergence minimizes the amount of executed instructions in case of branching, as in case of branching on a streaming multiprocessor, all instructions are executed anyway as the compiler uses branch predication to execute all paths of the branch. This can be mitigated by making branches dependent on thread IDs or by executing branching paths in different warps.

2.3.6 Algorithmic optimizations

Some algorithmic optimizations could be implemented in order to increase the level of parallelism or decrease the problem size. An example of this is deciding which landmarks to remove when the pool of landmarks is full. Removing landmarks that are old and lie close to the upper-left corner of the covariance matrix takes longer than removing newer landmarks that have columns and rows associated in the covariance matrix with higher indices. This problem is of course highly dependent on the implementation of the landmark removal and can be omitted by using pointers to the data represented in the matrix. However, changing the algorithm that selects landmarks for removal to prioritize newer landmarks might lower the average execution time of the landmark removal. Simultaneously, this increases the accuracy of the SLAM output when old landmarks are revisited, as these landmarks usually have a lower uncertainty than newer landmarks that have otherwise replaced these old landmarks.

2.3.7 Compiler options

As described by Oshana (2013), compiler optimizations can be used to further accelerate the implemented algorithms. The compiler for host code can be configured to maximize execution speed of the program by using global optimization methods. This way, the compiler has a complete overview of the program and does not need to make assumptions about external functions, leading to a better compilation of instructions.

3 Problem Analysis

3.1 Introduction

This chapter describes the problem analysis which serves three goals:

- mapping the bottlenecks of the ES-EKF SLAM algorithm and its software implementation as proposed by Shah (2020),
- placing the ES-EKF SLAM algorithm within the context of HAPI,
- constructing the qualitative and quantitative requirements of the accelerated ES-EKF SLAM implementation.

Mapping the bottlenecks of the ES-EKF SLAM algorithm assesses the problem regarding temporal optimization of the ES-EKF SLAM algorithm and helps finding opportunities for accelerating the algorithm; the operations that take the most time to execute are generally where a high speedup can be accomplished when these operations are accelerated. The mapping of these bottlenecks is reached through an algorithm complexity analysis, which theoretically analyzes the worst-case time complexity of the algorithm summarized in Algorithm 2.2 and Algorithm 2.3 and also provides a runtime analysis of the existing software implementation of the ES-EKF SLAM algorithm.

The context analysis places the ES-EKF SLAM algorithm within the context of HAPI in order to describe the functionality of the ES-EKF SLAM algorithm for the HAPI. This then leads to which requirements the ES-EKF SLAM implementation needs to fulfill to be useful for HAPI.

This chapter is concluded by a problem statement which states the prioritized system requirements. This will then provide a basis for the construction of the quantitative requirements that the accelerated ES-EKF SLAM implementation will fulfill during this project.

3.2 The bottlenecks of ES-EKF SLAM

This section maps the parts of the ES-EKF SLAM algorithm that take longest to in order to find opportunities for accelerating the execution time of the algorithm. The goal is to find the computations that can be optimized in order to provide the largest speedup of the algorithm. These computations are focused during the design of an accelerated implementation of the algorithm.

This is done in two parts: a theoretical worst-case time complexity analysis of the algorithm displayed in Algorithm 2.2 and Algorithm 2.3 is executed, and a runtime analysis of the existing software implementation is performed. The worst-case time complexity analysis expresses the computational time complexity of the ES-EKF SLAM algorithm using the big-O notation in order to assess the dependency of the execution time on its input. Additional to the ES-EKF computations, the runtime analysis covers the software implementation of the *bookkeeping* and landmark association algorithms, as these can not be covered during the worst-case time complexity analysis.

3.2.1 Worst-case time complexity of ES-EKF SLAM

First, the worst-case time complexity of the ES-EKF SLAM algorithm is assessed by expressing the complexity of the algorithm using the big-O notation. By expressing the computational complexity of each step in Algorithm 2.2 and Algorithm 2.3, the steps of which the execution time is theoretically most dependent on its input size can be found. As explained above, these steps offer opportunities for accelerating the ES-EKF SLAM algorithm.

3.2.1.1 Method

The worst-case time complexity analysis of the ES-EKF SLAM algorithm only covers the *update* and *prediction* steps of the ES-EKF SLAM algorithm, summarized in Algorithm 2.2 and Algorithm 2.3, as the *bookkeeping* algorithm that is implemented at the time of analysis, might still be changed and is not the main focus of this project. This analysis makes use of the following assumptions:

- The worst-case time complexity of multiplying the matrices of size $x \times y$ and $y \times z$ is $O(xyz)$ (Skiena, 2008).
- The worst-case time complexity of multiplying the vector of size y with a matrix of size $x \times y$ is $O(xy)$. This is derived from the worst-case time complexity of matrix-matrix multiplication, as a vector can be seen as a $1 \times y$ matrix.
- The worst-case time complexity of transposing a matrix of size $x \times y$ is $O(xy)$ (Shanker & Shameer, 2016).
- The worst-case time complexity of inversing a matrix of size $x \times x$ by Gauss-Jordan elimination is $O(x^3)$ (Sharma, Agarwala, & Bhattacharya, 2013).
- Since $L \leq K$, the worst-case exists when $L = K$, so it is assumed that this is the case.

3.2.1.2 Results

Equation	Worst-case time complexity
1	$O(L)$
2	$O(L)$
3	$O(L)$
4	$O(L)$
5	$O(LK^2)$
6	$O(L^3)$
7	$O(LK)$
8	$O(K)$
9	$O(LK^2)$
10	$O(1)$
11	$O(1)$
12	$O(1)$
13	$O(1)$
14	$O(1)$
15	$O(K)$

Table 3.1: Computational complexity of each step of the ES-EKF SLAM algorithm as listed in Algorithm 2.2 and Algorithm 2.3.

Table 3.1 provides an overview of the worst-case time complexity of the ES-EKF algorithm per Equation as listed in Algorithm 2.2 and Algorithm 2.3. It can be observed that the time complexity of the *update* step of the EKF-SLAM algorithm is $O(LK^2)$. This is confirmed by Thrun, which explains that the quadratic complexity of EKF SLAM stems from the matrix-matrix multiplication in the *update* step (Thrun, Burgard, & Fox, Probabilistic Robotics, 2006). L is smaller than K, or, in the worst-case L is equal to K, as L is the number of observed landmarks and K is the number of landmarks in the pool. This makes the worst-case time complexity of the *update* step of the ES-EKF SLAM algorithm $O(K^3)$. This means that Equation 5, 6, and 9 take longer to compute when the number of landmarks in the pool increases with relation to the other Equations.

Furthermore, as Equations 10 to 14 only involve the pose of the camera device, no calculations involving the landmarks are performed during the *prediction* step of the algorithm. These Equations can therefore be simplified as follows:

The computation of $f\left(\breve{x}(t|t)\right)$ according to Equation 11 in Algorithm 2.3 is simplified as:

$$f(\breve{x}(t|t)) = \begin{bmatrix} I_{3\times3} & 0_{3\times4} & R_w\Delta & 0_{3\times3} \\ 0_{4\times3} & \exp(W\Delta) & 0_{4\times3} & 0_{4\times3} \\ 0_{3\times3} & 0_{3\times4} & \Gamma & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times4} & 0_{3\times3} & \Lambda \end{bmatrix} \begin{bmatrix} p^w(t) \\ q^w(t) \\ v^c_{wc}(t) \\ \omega^c_{wc}(t) \end{bmatrix}$$

And therefore, the computation of $\breve{x}(t+1|t)$ becomes:

$$\breve{x}(t+1|t)(1{:}12) = f(\breve{x}(t|t))$$

And finally, the computation of $F(\hat{x}(t|t))$ can be simplified as:

$$F(\hat{x}(t|t)) = \begin{bmatrix} I_{3\times3} & P_\theta & R_w\Delta & 0_{3\times3} \\ 0_{3\times3} & I_{3\times3} & 0_{3\times3} & \frac{1}{2}I_{3\times3}\Delta \\ 0_{3\times3} & 0_{3\times3} & \Gamma & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & \Lambda \end{bmatrix}$$

With this simplification, Equation 15 needs to be adjusted accordingly and can be computed in three steps, reducing the complexity of the matrix-matrix multiplication to $O(k)$ as:

$$C(t+1|t)(1{:}12,12{:}12) = FC(t|t)(1{:}12,1{:}12))F^T + C_w$$

$$C(t+1|t)(1{:}12,13{:}12+k) = FC(t|t)(1{:}12,13{:}12+k))$$

$$C(t+1|t)(13{:}12+k,1{:}12) = C(t|t)(13{:}12+k,1{:}12))F^T$$

The size of the input to these Equations is therefore constant, making the time complexity of Equation 10-14 $O(1)$ and the complexity of Equation 15 $O(K)$. This means that the computational complexity of the *prediction* step of the EKF SLAM algorithm is $O(K)$.

As the *update* step has the highest order complexity within the algorithm, the overall worst-case time complexity of the ES-EKF SLAM algorithm becomes $O(K^3)$. This means that the execution time of the algorithm is highly dependent on the number of landmarks being tracked.

3.2.1.3 Conclusion of the worst-case time complexity analysis

As discussed above, the worst-case time complexity of the ES-EKF SLAM algorithm is $O(K^3)$, with the individual complexity of the *update* step summarized in Algorithm 2.2 being $O(K^3)$, and the complexity of the *predict* step summarized in Algorithm 2.3 is $O(K)$. It can thus be concluded that the *update* step of the algorithm has the highest worst-case time complexity. Diving deeper into the *update* step, it can be concluded that Equation 5, 6, and 9 of Algorithm 2.2 have the highest order time complexity and are thus expected to have an execution time that is highly dependent on the number of landmarks K and the number of measurements used for the *update* step L.

3.2.2 Runtime analysis of ES-EKF SLAM

This section discusses the runtime analysis of the of the ES-EKF SLAM algorithm implemented in MATLAB by Shah and Van der Heijden. The runtime analysis is performed by running the algorithm and analyzing the execution time of the different steps of the algorithm using the profiler. This results in the execution times of the different steps of the algorithm as implemented without any optimizations. These results are then used to confirm the findings from the worst-case time complexity analysis and to map the bottlenecks of the *bookkeeping* and landmark association steps of the algorithm that was not discussed during the worst-case time complexity analysis.

3.2.2.1 Method

In order to measure the execution time of the ES-EKF SLAM algorithm, the algorithm is executed in MATLAB 2019b on an Intel Core I7 6700K, which is clocked at 3.98GHz. The maximum number of landmarks in the pool is 1000 and the maximum number of landmarks that are added to the pool is 20. At least 5 landmarks are removed to make space for at least 5 new landmarks at every timestep. In total, 549 measurements are taken, with a measurement period of 0.1 second.

The MATLAB profiler collects the total execution times of each step in the algorithm and the resulting execution times are then used to determine the steps that take longest to execute. The steps of the algorithm that are of interest are the steps described in Algorithm 2.2 and Algorithm 2.3, and the steps that in total take over a second to compute over all iterations.

3.2.2.2 Results

The results of the runtime analysis are divided in two parts. First, the execution times of the steps described by Algorithm 2.2 and Algorithm 2.3 are discussed. Second, the steps of the *bookkeeping* algorithm that in total take over a second to execute are discussed. This results in a list of steps in the algorithm with a relatively high execution time.

The total execution time of the ES-EKF SLAM algorithm is 105.363 seconds, 99.078 seconds of which was spent in computations within the state *update* and state *predict* steps or in the *bookkeeping* step in computations that took over a second.

3.2.2.2.1 Execution time of the ES-EKF steps

Equation	Execution time (s)	% of total time
9	21.110	20.035
6	19.645	18.645
5	15.021	14.256
1, 2, 3 (combined)	0.778	0.738
10, 11, 12 (combined)	0.183	0.174
8	0.158	0.150
7	0.128	0.121
13,14 (combined)	0.100	0.095
15	0.018	0.017
4	0.002	0.002

Table 3.2: Profiler results of the ES-EKF SLAM algorithm summarized in Algorithm 2.2 and Algorithm 2.3.

From the profiler analysis, it becomes clear that the *update* step is the most temporal expensive part of the SLAM algorithm. As can be seen in Table 3.2, Equation 9 takes the longest to compute, followed by Equation 6, and Equation 5. This confirms the findings in the worst-case time complexity analysis, as the time complexity of these Equations is $O(LK^2)$.

3.2.2.2.2 Bottlenecks in the bookkeeping algorithm

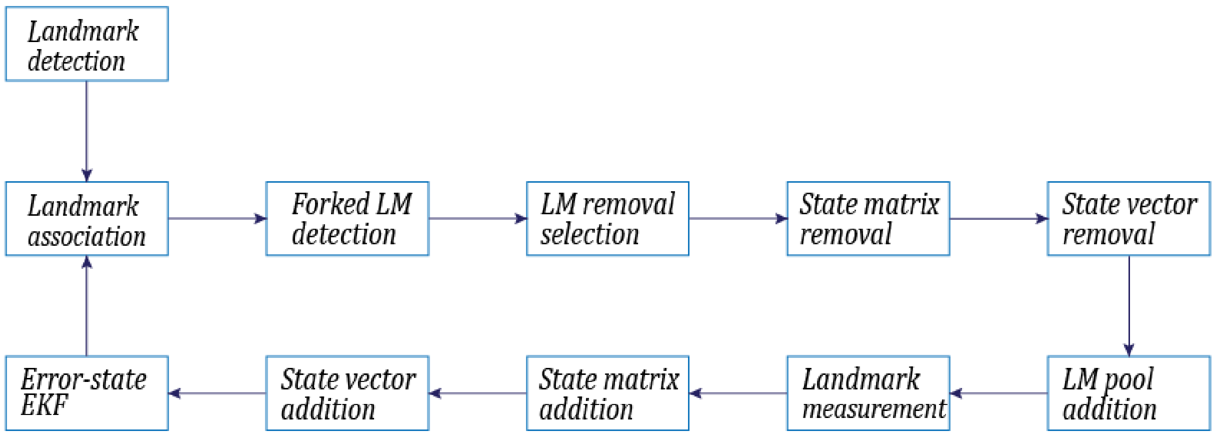


Figure 3.1: Diagram showing the steps of the *bookkeeping* algorithm.

Line	Operation	Execution time (s)	% of total time
1	Remove rows of deleted landmarks from C	11.077	10.513
2	Remove columns of deleted landmarks from C	10.653	10.111
3	Set the submatrix of the old landmarks	9.127	8.662
4	Descriptor matching for landmark association	5.438	5.161
5	Reserve memory for new error state covariance matrix with space for new landmarks	2.827	2.683
6	Set the new error state covariance matrix	2.822	2.678

Table 3.3 Lines and execution times of additional operations in the algorithm that take over a second to compute.

Other computations that take over a second to compute, which are not part of the EKF-SLAM algorithm, but are of importance in the *bookkeeping* algorithm shown in Figure 3.1, are shown in Table 3.3. Lines 1 and 2 remove selected landmarks from the state covariance matrix. Line 3 compares and matches landmarks in the pool with measured landmarks. Line 4, 5 and 6 are part of the algorithm where new landmarks are added to the covariance matrix. Line 4 initializes a placeholder matrix the size of $C(t|t)$, which is then filled diagonally in line 5. Line 6 copies the filled placeholder matrix into the object holding $C(t|t)$.

3.2.2.3 Conclusion of the runtime analysis

As discussed, the total execution time of 549 iterations of ES-EKF SLAM is 105.363 seconds. Most of this time was spent on executing the steps that regard the error state covariance matrix. These are the execution of Equations 6, 5, and 9, and the matrix mutations during the *bookkeeping* algorithm where landmarks are removed from the matrix and new landmarks are added. Furthermore, it can be concluded that the execution time of the descriptor matching for landmark association is relatively high.

3.2.3 Conclusion

The results of the runtime analysis support the theoretical computational complexity analysis. It can be seen that the steps with the highest order computational complexity take the longest to compute and are highly dependent on the size of the input, namely the number of landmarks in the pool and observed landmarks. These steps are the computation of Equation 5, 6, and 9, which contain matrix operations, including matrix-matrix multiplications, transposition, and inversion of potentially large matrices, dependent on the number of measured landmarks (L) and landmarks in the pool (K).

The results of the runtime analysis have shown that additional operations which are part of the *bookkeeping* algorithm are also computationally expensive. Most of these operations involve the error state covariance matrix $C(t|t)$ as well, which is a large matrix depending on the number of landmarks in the pool. The results additionally show that the algorithm used to compare the measured landmarks to the landmarks in the pool is also computationally expensive. As the total execution time of 549 ES-EKF SLAM iterations is 105.4 seconds, the ES-EKF computations and the *bookkeeping* steps that took in total over a second to compute took in total 99.078 seconds to execute.

3.3 Context analysis

In this section the context of the ES-EKF SLAM algorithm with regards to its application in HAPI is analyzed. This is done by describing how the ES-EKF SLAM algorithm is used for handheld perfusion imaging, what the functionality of the HAPI is, and discussing improvements of the ES-EKF SLAM algorithm based existing work on ES-EKF SLAM by Shah (2020). The goal of this section is to provide a description of requirements that the ES-EKF SLAM implementation needs to fulfill in order to be functional for handheld perfusion imaging.

3.3.1 ES-EKF SLAM for HAPI

The HAPI uses laser speckle contrast imaging to measure blood perfusion under the skin surface. The aim of this device is to realize enhanced reality rendering of tissue perfusion in order to visualize the data for medical users and to perform clinical studies in the area of dermatology, burn wound care and the diabetic foot.

There are two ways Visual SLAM is useful for handheld perfusion measurement: motion measurement and visualization of the measurement in enhanced reality. As explained in Chapter 1Visual SLAM can be used to measure the angular and translational velocity of the handheld measurement device in order to compensate the laser speckle contrast measurement for motion artifacts introduced by the operator and the person being measured. SLAM can also be used to create a sparse map of the measured surface in order to perform 3D reconstruction of the surface for projection of the perfusion map.

Enhanced reality rendering of the tissue perfusion includes projecting the perfusion images on the measured surface in 3D. This can either be done by the means of projecting the data on the skin surface itself (using Augmented Reality displays) or a projector. The projection can then be done in real-time. Virtual Reality can be used as well to project the perfusion map on a 3D model, which can be visualized after the measurement. For all cases, a 3D map of the measured skin surface is necessary to perform the projection, which can be generated by mapping landmarks on the skin surface by the means of Visual SLAM.

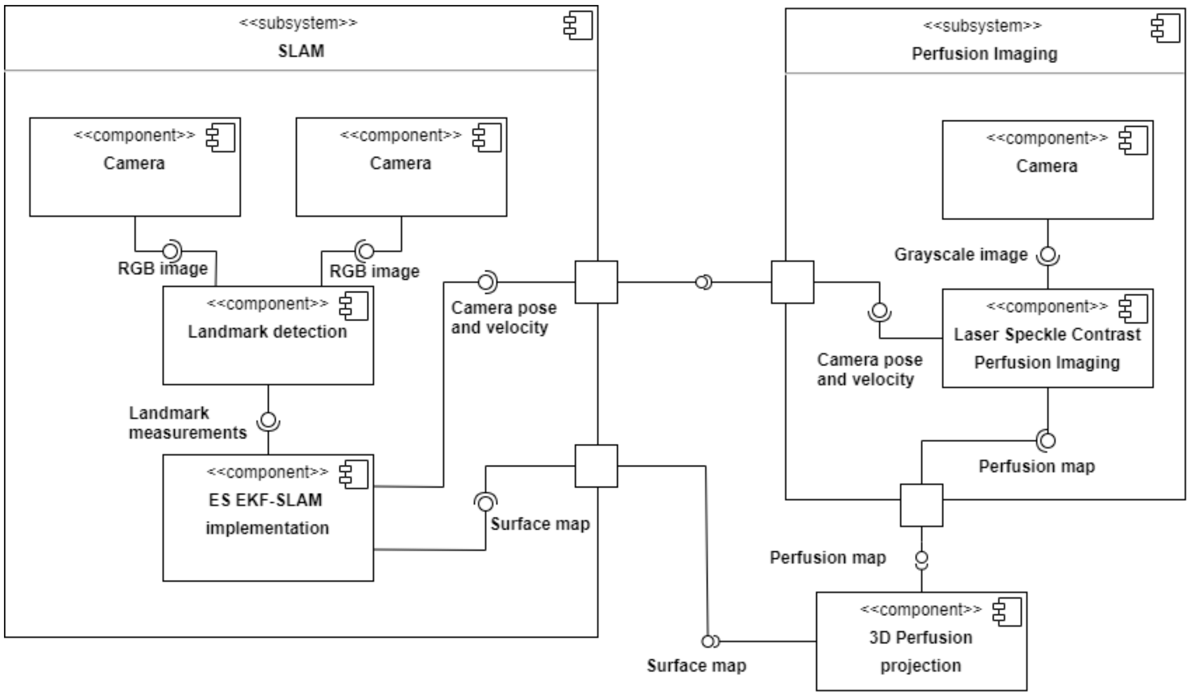


Figure 3.2: Component diagram of the HAPI System with SLAM integrated.

Figure 3.2 shows a component diagram of the HAPI with integrated Visual ES-EKF SLAM. The diagram shows that the surface map resulting from the SLAM algorithm is used for visualization and the camera pose and velocity are used for compensation of the laser speckle contrast image for motion artifacts.

3.3.2 The functionality of the HAPI

The Handheld perfusion imaging device is required to have the following features (Steenbergen, 2020):

- The HAPI should allow for handheld use, and therefore should be compact.
- The HAPI should enable perfusion imaging in extended areas of tissue.
- The HAPI should be robust against tissue curvature and movement artefacts.
- The HAPI should provide both the perfusion image and a high-quality color image.
- The HAPI should allow for analyzing tissue development in real time without using artificial markers.

Handheld use of the HAPI Device means that the measurement system must be wireless and battery powered. Furthermore, the handheld device must be compact and light enough to operate using at most two hands, without the introduction of additional motion artifacts due to weight. It was decided that the handheld device must be able to operate on the battery for at least two hours.

All features other than the compactness and handheld use of the device can be implemented using Laser Speckle Contrast Imaging, combined with Visual SLAM. The system must therefore be able to support both. However, this research project focusses on the implementation of the Visual ES-EKF SLAM algorithm, while leaving room and resources on the resulting system for implementation of the perfusion algorithm.

Finally, for the perfusion imaging it is important to note that a mono camera is used to capture the laser speckle greyscale images with a resolution of 2048 by 1536 pixels with a framerate of 40 frames per second. The duration of a measurement is 7 seconds. As discussed above, motion data resulting from the Visual SLAM algorithm is used to measure motion artifacts in order to compute the perfusion map. Therefore, it is important that the rate of motion measurement is high enough for accurate compensation. This rate is least 40 Hz.

3.3.3 Suggested improvements

Shah (2020) proposed fusing the EKF-SLAM algorithm with a gyroscope in order to track the pose of the camera system more accurately. This can also be used to increase the measurement rate of the rotational velocity. Additionally, an inertial measurement unit can be used for this purpose to measure the linear acceleration of the handheld measurement device in order to estimate the velocity with a larger measurement rate (van der Heijden, Visual Navigation, 2018).

As suggested by Shah (2020) as well, the Visual SLAM implementation will support a stereo-camera setup, as proposed by Shah (2020) and Grimm (2020). This setup uses two FriendlyElec cam1320 camera modules, each with a resolution of 4224 x 3136 pixels. Because the feature extraction algorithm used in the proposed algorithm uses greyscale images, there is no need for 3-channel RGB images. The proposed cameras are able to capture images with a rate of 30 frames per second but are also able to transmit images with a rate of 60 frames per second at a lower resolution.

3.4 Conclusion and problem statement

This section concludes the problem analysis by outlining the challenges of developing an accelerated implementation of the ES-EKF SLAM algorithm for the HAPI and the goals that need to be reached in order for ES-EKF SLAM to be functional for HAPI. The challenges arise from the algorithm complexity analysis and the context analysis which need to be overcome in order to reach the goals that are being set. Finally, the goals are defined as qualitative and quantitative requirements that the accelerated ES-EKF SLAM algorithm needs to fulfill in order to be effective for handheld perfusion imaging.

3.4.1 Challenges of designing an accelerated ES-EKF SLAM implementation for HAPI

The computational complexity analysis showed that the execution time highly depends on the number of landmarks in the pool and the number of observed landmarks at each timestep. Unfortunately, in order to provide a good 3D surface reconstruction, the system is required to generate a 3D map which is as accurate and dense as possible. This density is dependent on the number of landmarks and thus shows that there is a tradeoff between density and computation time. Additionally, a large number of landmarks also requires a large amount of memory, which increases exponentially for the state covariance matrix with the number of landmarks in the pool. Since the maximum number of landmarks during the runtime analysis in Section 3.2is set to 1000 while the rate of execution is 5.2 Hz, the minimum pool size is therefore required to be 1000 landmarks in order to provide an indication about the speedup of the accelerated ES-EKF SLAM implementation with regards to the MATLAB software implementation.

Not only the ES-EKF part of the SLAM algorithm suffers from dependency on the state size. The runtime analysis also showed that different parts of the *bookkeeping* algorithm are computationally expensive. Mutations on the state covariance matrix for adding and removing landmarks from the state and the algorithm to compare measured landmarks to landmarks in the pool take too much time to compute for the algorithm to perform in real-time with a high measurement rate in the current software implementation.

The rate of execution of the ES-EKF SLAM algorithm is required to be at least 10 Hz and can be executed in soft real-time in order to keep the processing time for analysis of the measurements as low as possible to enable fast measurement of a skin surface. On the other hand, the execution of the measurement acquisition is preferred to be hard real-time in order to perform accurate motion estimation. This is because the ES-EKF SLAM algorithm is also used for motion compensation of the perfusion data such that the resulting perfusion data can be visualized while taking the measurement. Therefore, the execution of landmark measurement acquisition needs to be executed with a rate of 10 Hz, while the execution of additional sensor measurement needs to be performed with a rate of 40 Hz to be functional for HAPI.

Since the HAPI is required to be wireless, lightweight, and compact; size and energy consumption of hardware is also of importance. This means that the computing device is required to be small and lightweight. The computing device is also required to be energy efficient in order to not require a large battery to keep the device operational for 2 hours.

Finally, the handheld perfusion imaging device is required to support both the laser speckle contrast imaging for perfusion imaging and the ES-EKF SLAM algorithm, as they need to be executed simultaneous within the same device. The handheld system should therefore support both the camera for handheld perfusion imaging and the camera system designed by Shah (2020) and Grimm (2020).

3.4.2 System requirements

In this section the construction of MoSCoW requirements is used to quantify the requirements of the accelerated ES-EKF SLAM implementation for HAPI. This list is then used to construct a list of functional requirements, UI requirements and design constraints that the designed ES-EKF SLAM implementation needs to fulfill and will act as guidelines throughout the design process.

3.4.2.1 MoSCoW requirements

The MoSCoW method for requirement construction helps with prioritizing tasks and setting constraints considering the finite development time. The resulting MoSCoW requirements are then used to construct functional requirements, UI requirements and design constraints and provide a rough outline of system requirements and constraints in a prioritized order.

Must	Should	Could	Will not
The system must allow real-time Visual SLAM with a minimum execution rate of 10Hz.	The system should be expandible for perfusion measurement.	The resulting system could visualize the Visual SLAM result in soft real-time.	The resulting system will not be optimized for other Visual SLAM algorithms than ES-EKF SLAM.
The measurement system must be handheld.	The resulting surface map should contain at least 1000 landmarks.	The resulting system could be operated handheld.	The resulting system will not be implemented on the HAPI platform during this project.
The measurement system must be wireless.			
The measurement rate of additional sensors should be at least 40Hz.			

Table 3.4: MoSCoW requirements of the accelerated ES-EKF SLAM implementation for HAPI.

The MosCoW requirements in Table 3.4 show that the main focus is put on the implementation of a Visual ES-EKF SLAM algorithm that supports landmark measurement acquisition with an execution rate of 10 iterations per second. The measurements are then used in the ES-EKF SLAM algorithm that must be able to process them with a similar execution rate of 10 Hz. The worst-case execution time of an ES-EKF SLAM iteration and landmark measurement acquisition is therefore required to be at most 100 ms while supporting a minimum map size of 1000 landmarks. Furthermore, as described in the context analysis, it is important that the landmark measurement device can be carried handheld and operate wirelessly.

As the ES-EKF SLAM algorithm should be implemented next to the perfusion imaging system, it is of importance that the platform leaves physical space and computing resources for the perfusion imaging system.

Functionalities that are not of high importance for the execution of the ES-EKF SLAM computations are the soft real-time visualization of the state of the system and the handheld operation of the measurement device. Furthermore, it is important to state that the resulting system will only implement the ES-EKF SLAM algorithm and that the system will not be implemented on the HAPI platform during this project, resulting in a purely functional implementation of the ES-EKF SLAM algorithm.

3.4.2.2 Functional requirements

Functional requirements specify the functional behavior of the accelerated ES-EKF SLAM implementation. This includes the specification of the required execution speed, system architecture requirements, and requirements of the ES-EKF SLAM results.

FR1. The system must allow soft real-time Visual SLAM with a minimum execution rate of 10Hz.
FR2. The landmark measurement acquisition rate must execute with a rate of at least 10 Hz.
FR3. The measurement rate of additional sensors should be at least 40Hz.
FR4. The measurement system must be handheld.
FR5. The measurement system must communicate wirelessly.
FR6. Additional sensors should be implemented for increased performance of the ES-EKF SLAM algorithm.
FR7. The system should be expandible for perfusion measurement.
FR8. The resulting surface map should contain at least 1000 landmarks.

3.4.2.3 UI requirements

The UI requirements include specifications of the user interface of the system with the implemented ES-EKF SLAM algorithm. This includes visualization of the ES-EKF SLAM results and the operation of the system.

UR1. The resulting system must contain an interface to start landmark measurement acquisition with a set of adjustable settings for image size and acceleration options.
UR2. The resulting system most contain an interface to start the receiving of measurements and execute the ES-EKF SLAM algorithm with a set of adjustable settings for maximum pool size, number of new landmarks per iteration, maximum number of landmarks used for the ES-EKF *update* and various acceleration options.
UR3. The resulting system could carry a viewer to visualize the 3D surface map in real-time.
UR4. The resulting system could be operated handheld to start and stop measurement acquisition and ES-EKF SLAM processing.

3.4.2.4 Design constraints

As described, design constraints are set due to the limited time for development. These constraints explain what will not be implemented or executed during the design and the development of the accelerated ES-EKF SLAM implementation.

DC1. The resulting system will not be optimized for other Visual SLAM algorithms than ES-EKF SLAM.
DC2. The measurement system will not include an embedded power source during this project.
DC3. The measurement system will not include integrated camera support for image acquisition during this project.

4 Design Space Exploration

This chapter describes the design space exploration that guides the decision-making process regarding the design of the system on which the ES-EKF SLAM algorithm will be implemented and how the algorithm will be accelerated. The goal is to provide a comparison between different computer architectures, system architectures, and hardware platforms for optimal execution of the most time expensive elements in the ES-EKF SLAM algorithm.

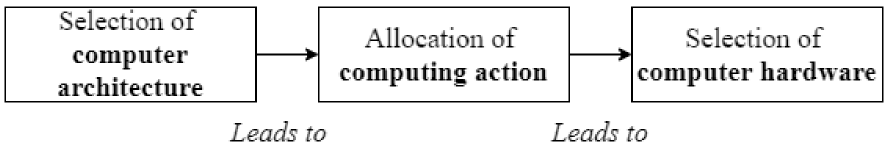

Selection of computer architecture
Allocation of computing action
Selection of computer hardware
Leads to
Leads to

Figure 4.1: Steps taking during this design space exploration.

Each step in this design space exploration leads to the next decision to be made, as shown in Figure 4.1. The first decision considers the computer architecture for the computation of the most time expensive part of the ES-EKF SLAM algorithm as concluded in Section 3.2, the matrix-matrix operations of the ES-EKF *update* algorithm. This leads to the second step where the computing actions are allocated on the system. This takes in account the functional requirements from Section 3.4 and results in a design for a computing system architecture. The third step takes this computing system architecture and presents a choice of computer hardware on which the accelerated ES-EKF SLAM algorithm is implemented.

During each step in this process the different options are scored on a scale of -2 to +2 against different metrics. Each metric has a weight, which is dependent on the importance of the metric for the decision. The resulting scores are then summed for each choice and the total scores are used to decide on which option to use from that step onward.

4.1 Choosing an Applicable Computer Architecture for ES-EKF SLAM update

As defined in Section 3.2, the computational bottleneck of ES-EKF SLAM are the matrix computations within the ES-EKF Update. Fortunately, matrix operation algorithms can be parallelized on many-core systems in order to decrease their execution time. In this section the use of many-core systems for matrix computation is compared in order to make a decision about the computer architecture on which the computationally expensive parts of the ES-EKF SLAM algorithm are executed. The use of Graphical Processing Units (GPU) is compared against the use of many-core architectures that are implemented on FPGAs. The GPU that implements CUDA cores and Tensor cores is discussed in Appendix 2.

4.1.1 Weighted metrics

As discussed above, each metric has different importance and it is thus given an individual weight. This section explains each metric into detail as well as its weight assigned.

4.1.1.1 Performance

The *performance* metric has a weight of 10 as it is the most important. The performance of the system defines the number of operations per second that can be achieved. This metric is of most importance for accelerating the ES-EKF SLAM algorithm and achieving an execution rate of 10 iterations per second with at least 1000 landmarks in the pool as specified in Section 3.4.

4.1.1.2 Flexibility

The *flexibility* metric has a weight of 6. *Flexibility* defines how easy expansion of the system is in order to increase functionality. The flexibility metric also defines how easy it is to change the existing functionality. For instance, changing the behavior of the SLAM algorithm or to fit other elements within the Handheld perfusion imaging device such as the speckle contrast imaging.

4.1.1.3 Map density

The *map density* metric has a weight of 7 and defines the number of landmarks that can be achieved with the design choice. This number should be as high as possible in order to provide a dense map but is required to be at least 1000 landmarks. Unfortunately, there is a tradeoff between map size and performance, and performance is more important in this application, therefore justifying the weight given.

4.1.1.4 Energy consumption

The *energy consumption* metric has a weight of 3 and defines the energy consumption per computation. The energy consumption varies per design choice and is not equally important for each element in the system. However, it is desirable to keep the overall energy consumption as low as possible. As described later in this chapter, the possibility to perform the filtering in the SLAM algorithm can be performed on a machine other than the handheld device. The *energy consumption* metric is therefore of low importance.

4.1.1.5 Development effort

The *development effort* metric has a weight of 5 and quantifies the amount of time needed to implement the design choice. As the development of a prototype of the implementation should fit within the required timeframe, this criterion is of importance to keep the complexity of the system as low as needed.

4.1.2 Scoring overview

Table 4.1 provides a scoring overview for each computer architecture option. The scores for each option are motivated in the following sections.

Architecture	Weight factor	CUDA Core	Tensor Core	FPGA
Performance	10	1	2	-1
Flexibility	6	1	1	0
Map Density	7	-1	-1	1
Energy Consumption	3	1	1	2
Development Effort	5	1	1	-2
Weighted total score	-	14	24	-7

Table 4.1 Applicable computer architecture score table

Counting the scores, it can be seen from Table 4.1 that performing the matrix computations on the GPU is more favorable. Specifically, the potential increase of runtime performance when matrix multiplications are performed on the Tensor core means that state of the art GPUs are the most optimal platform to develop a fast implementation of the ES-EKF SLAM algorithm.

The GPU can also provide acceleration of parallelized algorithms such as feature finding and descriptor matching for landmark measurement acquisition. This means that the system that performs the ES-EKF SLAM computations and the device that performs the landmark measurement acquisition can both use a GPU in order to accelerate the computations.

4.1.3 Performance scoring

Based on Section 2.2 the FPGA receives a *performance* score of -1, the CUDA core receives a score of 1 and the Tensor core receives the highest score of 2.

Section 2.2 provides a comparison of the performance of BLAS operations on GPUs with CUDA cores, GPUs with Tensor cores and FPGA implementations that aim at accelerating the execution rate of matrix-matrix operations. This provides a good baseline for the comparison of performance of these different computer architectures as the most time expensive parts of the ES-EKF SLAM algorithm consist of matrix-matrix operations.

As discussed in Section 2.2, GPUs using CUDA cores have a higher runtime performance of dense BLAS operations compared to various FPGA implementations. This gives the FPGA a score of -1 and the GPU CUDA core a score of 1 for *performance*.

On the other hand, works discussed in this Section 2.2 also conclude that FPGAs have a better runtime performance of spare BLAS operations. However, since sparse matrix computations are a rather small part of the ES-EKF SLAM *update*, this has no major influence on the scoring.

As discussed in Section 2.2, newer generations of NVIDIA GPUs introduce the Tensor core, a computing unit that is specialized for performing matrix-matrix multiplications. As discussed this specialization ensures a large increase of matrix-matrix multiplications compared to its execution on the CUDA core. The Tensor core therefore has a *performance* score of 2.

As the state-of-the-art NVIDIA GPUs host both CUDA cores and Tensor cores, both type of architectures are from now on scored equally.

4.1.4 Flexibility scoring

The FPGA has a *flexibility* score of 0, while the GPU has a *flexibility* score of 1.

The scoring of the FPGA is justified by a tradeoff between runtime performance, flexibility, and re-configuration speed as described by Teubner & Woods (2013) in the context of FPGA development. Teubner & Woods (2013) define flexibility as the width of the range of problems that are solved using the FPGA implementation, while re-configuration speed defines the time needed to reconfigure the implementation for a new functionality. In this case, these two characteristics are grouped into one characteristic. As discussed above, a higher emphasis is put on the runtime performance of the SLAM algorithm, meaning that the flexibility of the system is sacrificed. Additional features can be executed on a general-purpose coprocessor, however, most processors offer a limited number of resources, of which many are being utilized in existing implementations that are described in Section 1.1. The FPGA implementation of the SLAM algorithm therefore has a score of 0.

GPUs are usually accompanied by a host general purpose CPU. As the GPU can host most of the computation for the ES-EKF algorithm, additional functionalities can be executed using the unutilized computing resources. Additionally, if needed, the parallel architecture on the GPU can be used for computations other than for Extended Kalman Filtering. For instance, the GPU can be utilized to also accelerate the bookkeeping algorithm. FPGAs on the other hand, require the implementation of a different computing unit to perform the time expensive landmark addition or removal. This gives the GPU implementation a score of 1.

4.1.5 Map Density scoring

The FPGA has a score of 1 for *map density* as its memory can be expanded, while the GPU has a score of -1 due to its fixed memory size.

If the runtime performance is not considered, the size of the map is limited by the amount of memory that is available on the system. Unfortunately, GPU’s have a fixed amount of fast on-board memory available, while the amount of memory can be expanded for FPGAs. For example, the HC-1 server card that is used by Jones, Powell, Bouganis, & Cheung (2010), which interfaces 128 GB of DDR2 RAM. This gives FPGAs an advantage regarding the number of landmarks that can be mapped and are therefore given a higher score of 1, while GPUs are given a lower and negative score of -1.

4.1.6 Energy Consumption scoring

The FPGA outperforms the GPU regarding energy consumption, giving the FPGA a favorable score of 2 against a negative score of 1 of the GPU.

It is well known that FPGA implementations can greatly reduce the energy consumption compared to general purpose processors, as described by Teubner & Woods (2013). This is confirmed by Nurvitadhi et. Al. (2017), who concluded that the FPGA has a higher performance per watt compared to a GPU for both dense and sparse matrix multiplication. This gives the FPGA an advantage over the GPU containing both Tensor cores and CUDA cores in terms of energy consumption. The FPGA therefore has a score of 2. On the other hand, GPUs provide a small energy advantage due to the fact that the high number of multiprocessors and lower clock rate decreases the energy consumption of computations performed on a GPU when compared to the computations performed on a CPU. This gives the GPU a lower score of 1.

4.1.7 Development Effort scoring

The GPU has a *development effort* score of 1, while the FPGA has a score of -2.

It is commonly known that the design process for FPGA applications is slow and more complex than regular software development. The development effort for FPGAs is therefore rather high compared to software development for GPUs as the Compute Unified Device Architecture (CUDA) programming model makes development for the CUDA supported GPU easy, as discussed in Appendix 2. This becomes more important if the fact that co-developers of the HAPI with no experience of FPGA development need to be able to implement the accelerated ES-EKF SLAM implementation on the HAPI is considered.

4.2 Allocation of computing action

After a system architecture has been defined in Chapter 4.1, a decision must be made regarding where the different computations of the algorithm are performed in this system. It is clear from the requirements in Section 3.4 that the measurement system on which the images are acquired is required to be handheld. As explained later in this section, handheld computers usually provide fewer computing resources than computers that are stationary and not handheld such as desktop computers or server computers.

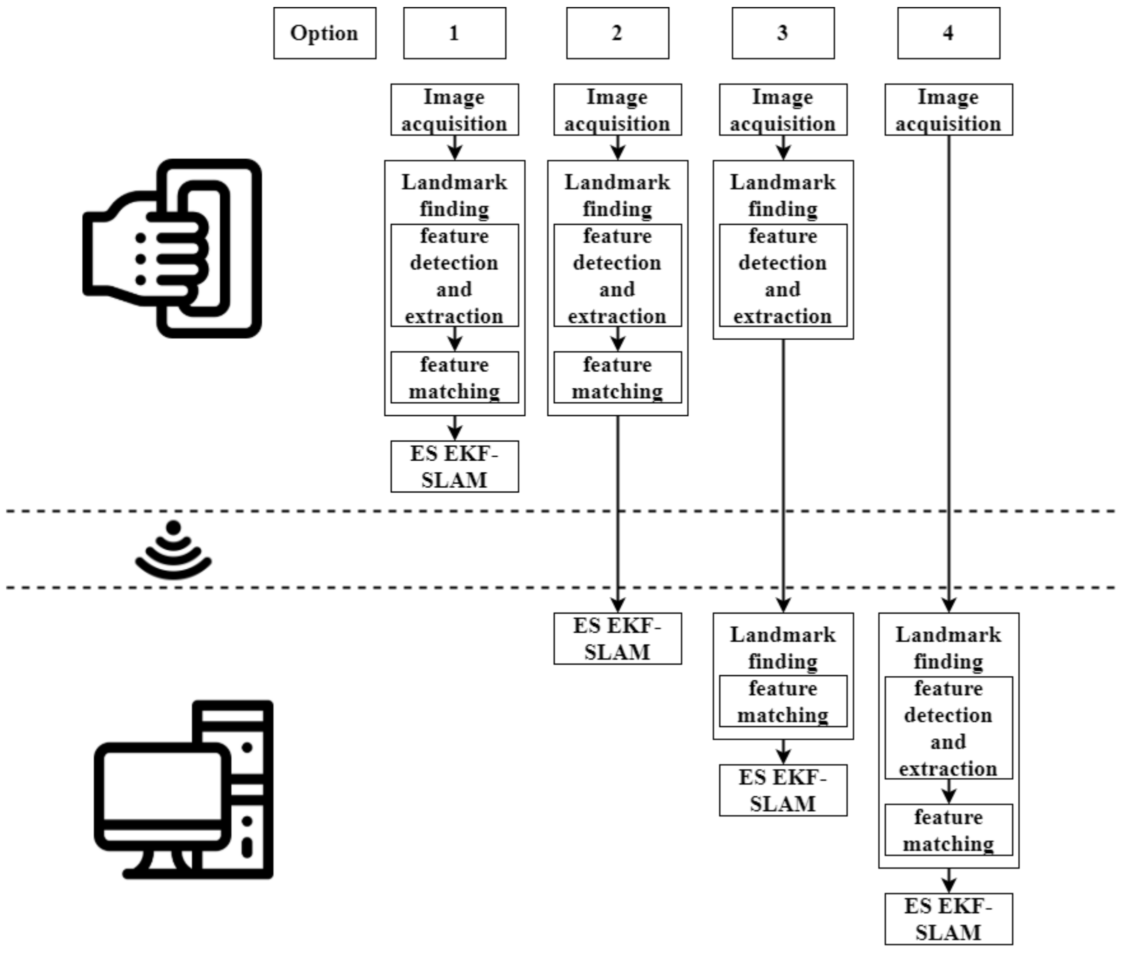


Figure 4.2: Different options for the allocation of computing action.

In order to increase available computing resources without sacrificing the handheld functionality can be performed by adding a central computer with more resources. This leads to four options on how the computations can be allocated over the different computing components in the system, as shown in Figure 4.2:

Option 1. All computations are performed on the handheld device.
Option 2. Image acquisition and landmark finding is performed on the handheld device and the ES-EKF SLAM computations are performed on a central computer.
Option 3. Image acquisition and feature finding for landmark finding is performed on the handheld device and the feature matching for landmark finding and the ES-EKF SLAM computations are performed on a central computer.
Option 4. Only image acquisition is performed on the handheld device, while image processing for landmark finding and the ES-EKF SLAM algorithm are performed on a central computer.

4.2.1 Weighted metrics

As discussed in Section 4.1, each metric is of different importance and therefore are given individual weights. This section explains each metric and the weight it receives. Compared to Section 4.1, the metrics of the *performance*, *flexibility*, *map density*, and *development effort* are similar and are weighted equally.

4.2.1.1 Performance

The *performance* metric has a weight of 10 as it is the most important. The performance of the system defines the number of operations per second that can be achieved. This metric is of most importance for accelerating the ES-EKF SLAM algorithm and achieving an execution rate of 10 landmark measurements and ES-EKF SLAM iterations per second with at least 1000 landmarks in the pool as specified in Section 3.4.

4.2.1.2 Flexibility

The *flexibility* metric has a weight of 6 and defines how easy expansion of the system is in order to increase functionality or change the existing functionality such as changing the behavior of the SLAM algorithm or to fit other elements within the Handheld perfusion imaging device such as the speckle contrast imaging.

4.2.1.3 Map density

The *map density* metric has a weight of 7 and defines the number of landmarks that can be achieved with the design choice. This number is required to be at least 1000 landmark as per requirement FR8. However, in order to provide a dense map, it is required to be as high as possible. Unfortunately, there is a tradeoff between map size and performance, and while performance is more important in this application, map density has a weight of 7.

4.2.1.4 Network traffic

The *network traffic* metric has a weight of 7 and describes the load that the system puts on the network for communication between the handheld device and a central computer. This load is desired to be very low, meaning that a lower amount of communication between system components results in a higher score.

4.2.1.5 Energy consumption

The *energy consumption* metric has a weight of 7 and defines the energy consumption per computation. In this case, the energy consumption of the handheld device is of importance as it is desired to be low in order for the implementation of the accelerated ES-EKF SLAM algorithm to not put a large load on the energy consumption when integrated on the HAPI.

4.2.1.6 Development effort

The *development effort* metric has a weight of 5 and quantifies the amount of time needed to implement the design choice. As the development of a prototype of the implementation should fit within the required timeframe, this criterion is of importance to keep the complexity of the system as low as needed.

4.2.2 Scoring overview

Table 4.2 shows the scoring of the different computing allocation choices. The scores for each option are motivated in the following sections.

	Weight Factor	Option 1	Option 2	Option 3	Option 4
Performance	10	-2	2	1	-1
Flexibility	6	-1	-1	1	2
Map Density	7	-2	2	1	0
Network traffic	7	2	1	0	-1
Energy consumption	7	1	0	-1	0
Development Effort	5	2	0	-1	1
Weighted total score	-	-9	35	11	0

Table 4.2 Allocation of computing score table

It can be seen from the table above that option 2 has the highest score as its distributed computing is expected to provide major performance benefits and will enable the densest map of the measured surface, while sacrificing energy consumption and flexibility. It makes therefore sense to develop a system where the “light-weight” landmark acquisition computation is performed on the handheld device, sending the landmark measurements to a desktop computer for execution of SLAM.

4.2.3 Performance scoring

Option 1 has a *performance* score of -2. Increasing the number of computing devices for landmark finding and ES-EKF SLAM computations leads to a higher *performance* score, as option 2 has a score of 2 and option 3 has a score of 1. Option 4 has a score of -1.

As the size of computing devices is needed to be relatively small compared to computers that are stationary, handheld computer devices provide fewer computing resources such as processors and memory than for example, desktop computers. Furthermore, the clock frequency of single-board computers for handheld devices is rather low to minimize energy consumption and heat generation. This leads to a large decrease in performance when computations are done on the handheld device compared to the execution of these computations on a central computer. Option 1 therefore has a score of -2, since this option requires the single board computer to perform both the computations for landmark finding and the ES-EKF SLAM computations.

Option 2 and option 3 make use of multiple computing devices for landmark finding and the ES-EKF SLAM computations. This allows for a pipelined implementation where landmark measurement acquisition and the ES-EKF SLAM computations can be parallelized, utilizing the computer on the handheld measurement device and the central computer.

Option 2 performs all parallelized computer vision algorithms on the handheld device, leaving the desktop GPU to use as many resources as possible for the execution of the EKF SLAM algorithm, while only reserving a small portion of resources for landmark matching. This is due to the fact that only a portion of the landmark measurements is selected, based on the landmark measurements with the largest strength. This positively impacts the performance, giving this option a score of 2.

Option 3, on the other hand, utilizes the desktop GPU for feature matching. As the GPU in the desktop computer is assumed to carry more SIMT processors and memory, it is more capable to perform parallelized image processing computations, while taking away some of the strain on the handheld device. Even though the desktop GPU is expected to perform faster matching of feature descriptors, this would decrease the level of pipelining, as the desktop GPU would utilize SIMT processors for descriptor matching of many features, while these processors would otherwise be used for EKF SLAM computations. This negatively impacts the performance of this solution compared to solution 2, which weighs against the fact that descriptor matching performs faster on the desktop GPU, giving this option a *performance* score of 1.

Option 4 does not make use of multiple computing devices in the system for landmark finding and the execution of the ES-EKF SLAM computations. This increases the number of operations to be executed on the central computer, leaving the computer in the handheld device unutilized. This decreases the number or resources that are available on the central computer for the ES-EKF SLAM computations as these are now used for landmark finding. This results in a *performance* score of -1 for this option.

4.2.4 Flexibility scoring

Option 1 and option 2 have a *flexibility* score of -1. Option 3 has a *flexibility* score of 1, while option 4 has the highest *flexibility* score of 2.

As discussed above, the handheld computing device is expected to carry little computing resources. As ES-EKF SLAM is a computationally expensive algorithm, it is expected that the implementation of the algorithm uses most of the resources available on the handheld measurement device. This means that there are fewer computing resources available to accommodate additional functionality such as the perfusion imaging algorithm. This gives option 1 a score of -1 for *flexibility*.

Furthermore, option 2 only makes landmark information available to the desktop computer. This decreases the amount of data available for other applications, such as visualization of the RGB images of the measured surface for later analysis, or the use of the data for other functionalities. On the other hand, the reduced bandwidth resulting from only the necessary landmark information being transmitted ensures that other data can be transmitted to the desktop computer. On desktop computer other functionalities can then be implemented to make use of this data. This way for instance, a small set of images can still be transmitted for further analysis by the operator. Unfortunately, this option requires more resources from the on-board GPU of the handheld device, limiting the utilization of the GPU for other functionalities that utilize the GPU. A graphical user interface has therefore limited possibilities to use the GPU if needed. This limits the flexibility of this option, giving it a score of -1.

Option 3 on the other hand leaves resources on the handheld device for additional functionalities, dependent on the needs of the operator, such as interfacing functionalities that make use of the on-board GPU. This leaves more possibilities to expand the functionality of the handheld device, increasing the flexibility of this solution. This gives this option a flexibility score of 1.

Since option 4 makes the raw images are available on the desktop computer, these images can be used for other functionalities, therefore increasing the flexibility of the system. This justifies the *flexibility* score of 2 for option 4.

4.2.5 Map density scoring

Option 1 has a score of -2 for *map density*, option 2 has a score of 2, option 3 has a score of 1 and option 4 receives a score of 0.

The score of option 1 is again motivated by the fact that a handheld computer provides fewer computing resources. In this case RAM is of high importance as a larger map requires more memory. Furthermore, a larger map size increase the computational complexity, as described in Chapter 3.2. This gives option 1 a score of -2 for *map density*.

Option 2 makes most use of the computing resources on the central computer for ES-EKF SLAM, meaning that it makes as much use as possible of the resources available for the execution of the EKF SLAM algorithm, while only reserving a small portion of resources for landmark matching. This is because only a subset of the landmark measurements is selected for transmission based on the landmark measurements with the largest strength. This results in a score of 2 for option 2.

Option 3 and option 4 also make use of the central computer for the ES-EKF SLAM computations. Both options, however, need to sacrifice resources for the landmark measurement algorithms, decreasing the amount of memory available for the ES-EKF SLAM computation. This negatively impacts the *map density* for both options. Option 3 receives a score of 1, as for this option only space is required for the feature descriptors to be matches, while option 4 receives a lower score of 0, as this option requires memory to be reserved for the input images.

4.2.6 Network traffic scoring

For the *network traffic* metric, option 1 receives a score of 2, option 2 receives a score of 1, option 3 receives a score of 0 and option 4 receives a score of -1.

Option 1 requires no transmission of large data objects such as images and landmark measurements. The resulting network traffic is therefore low, giving this option a score of 2 for *network traffic*.

Option 2 receives a slightly lower score, as it uses the wireless network to transmit data, but much less compared to option 3 and option 4, as this option performs the landmark acquisition on the handheld device where the features are discovered and matched, and only a set of the strongest matches is transmitted to the desktop device. As only the data relative to the strongest landmarks needs to be transmitted, network traffic is moderately low, giving this option a score of 1.

Option 3 increases the amount of data transmitted wirelessly, as the descriptor data of all features in both images is needed, the amount of data being transmitted for each ES-EKF SLAM iteration. Considering that an image may contain more features than landmark measurements are needed, a lot of redundant data is transmitted. This gives option 3 a network traffic score of 0.

Option 4 requires the most network usage, as for this option wirelessly transmits a stereo image pair. This is much more data than a set of descriptors, or a subset of landmark measurements. This gives this option a negative score of -1.

4.2.7 Energy consumption scoring

For *energy consumption*, option 1 receives a score of 1, option 2 receives a score of 0, option 3 receives a score of -1 and option 4 receives a score of 0.

As radios often require a lot of energy to transmit data, computing a byte of data requires less energy than transmitting a byte, the fact option 1 requires very little use of the wireless transceiver to transmit large data objects such as images and landmark measurements favors the scoring of option 1. Even though the handheld device needs to perform all computations on-board, this option receives a score of 1, as network traffic is low and little energy is used to transmit data.

Option 2 makes more use of a wireless transceiver to transmit data than option 1, next to performing the computations for landmark measurement acquisition. This increases the energy consumption, even though the amount of transmitted data is low compared to option 3 and option 4. Therefore option 2 receives a score of 0 for *energy consumption*.

Option 3 makes even more use of the wireless transceiver than option 2, as more data needs to be transmitted regarding the landmark measurements. This option therefore has an increased energy consumption of the wireless communication, while still performing the computations for feature detection and descriptor extraction. This increases the energy consumption. Also considering that an image may contain more features than landmark measurements are needed, a lot of redundant data is transmitted, wasting the energy needed for data transmission. Taking this in consideration, option 3 receives a score of -1 for *energy consumption*.

As discussed, option 4 requires the handheld device to wirelessly send the stereo image data to the central computer. Wirelessly transmitting the images likely consumes more energy than processing the images and transmitting the results, which will result in less use of the wireless transceiver that consumes more energy than the processor. On the other hand, no image processing is performed on the handheld device, resulting in less power consumption of the on-board computer. This, however, only has a small impact on the energy consumption of the handheld device, resulting in a score of 0.

4.2.8 Development effort scoring

For *development effort*, option 1 has a score of 2, option 2 has a score of 0, option 3 has a score of -1 and option 4 has a score of 1.

Development effort for option 1 is moderately low, as all operations are performed within the handheld device. Therefore, no additional infrastructure is needed, giving the *development effort* of option 1 a score of 2. Option 2, 3, and 4 have an increased development effort, due to the introduction of a wireless network infrastructure. Option 2 requires computer vision algorithms to be implemented on both the handheld device and the desktop computer (for landmark matching). However, as these algorithms are implemented similarly on both devices, development effort is not impacted negatively nor positively by this. This results in a score of 0 for option 2. Option 3 however, adds the fact that the allocation and management of resources needed for descriptor matching on the central computer need to be taken in account. Furthermore, the network infrastructure needs to support the transmission of large data objects, negatively impacting the development effort. This results in a score of -1 for option 3. Option 4 also needs to take in account the allocation and management of resources needed for landmark measurement acquisition, development effort is less impacted by this option. This is because only a network infrastructure for wireless image streaming needs to be implemented. The execution of both the ES-EKF SLAM computations and landmark measurement acquisition are implemented on the central computer, requiring less development on a separate device. This results in a development effort score of 1 for option 4.

4.3 Selection of computer hardware

Now that a computer architecture is chosen for the acceleration of the ES-EKF SLAM algorithm, and a decision has been made on the allocation of the computations, the computer hardware must be selected. This mainly consists of the selection of the single board computer on which the handheld landmark measurement acquisition is implemented, and the GPU that will be placed in the central computer. This selection is limited to devices with NVIDIA GPUs that are CUDA compatible, in order to constrict the choices and to limit development effort of the implementation on the handheld device and the central computer.

4.3.1 Weighted metrics

As in the sections above, each metric is of different importance. For the hardware selection, different metrics apply and each metric is assigned a different weight. The metrics are defined and weighted below.

4.3.1.1 CUDA cores

The *CUDA cores* metric has a weight of 8 and specifies the number of threads that can be executed simultaneously. A higher number of CUDA cores thus increases potential parallelism, therefore increasing the runtime performance of parallel algorithms.

4.3.1.2 Tensor cores

The *Tensor cores* metric has a weight of 10 and specifies the number of Tensor cores a GPU has that can be used to accelerate the matrix-matrix multiplications of ES-EKF SLAM. This metric therefore only applies to the GPU on the central computer.

4.3.1.3 Memory

The *memory* metric has a weight of 7. The amount of memory available on a GPU influences the map size. The GPU needs to provide enough memory to store the data of the ES-EKF SLAM computations in order to fulfill the minimum map density requirement of 1000 landmarks. This justifies the assigned weight.

4.3.1.4 Energy consumption

The *energy consumption* metric has a weight of 7. This metric only applies to the handheld device as this device needs to be energy efficient enough to function handheld and wirelessly.

4.3.1.5 Availability

The *availability* of the computer hardware has a weight of 6. This metric is of importance as it represents the time between ordering and receiving of the hardware, which is required to be low due to the limited time for development during this project.

4.3.1.6 Cost

The *cost* metric has a weight of 5. The cost of the computer hardware is desired to be as low as possible in order for it to be affordable during this project and future projects where this system may be duplicated. As the budget during this project is relatively large, the cost is not highly significant.

4.3.2 The handheld computing device

	NVIDIA Jetson Nano	NVIDIA Jetson TX2	NVIDIA Jetson Xavier NX
CUDA Cores	128	256	384
Tensor Cores	0	-	48
CPU	Quad-core ARM Cortex -A57	Dual-core Denver 2 and quad-core ARM Cortex -A57	6-core NVIDIA Carmel ARM v8.2
RAM (GB)	4	8	8
Power (W)	5/10	7.5/15	10/15
Price (€)	105	406.99	419

Table 4.3: Computer hardware options for the handheld device.

This section assigns each development board shown in Table 4.3 a score according to the metrics described above. The table shows the selection of development boards containing single board computers with an NVIDIA CUDA compatible GPU. The development boards differ in price, computing resources and energy consumption. Additionally, the development boards must be able to interface the camera system designed by Shah (2020) and Grimm (2020). All selected development boards therefore provide two CSI ports for camera interfacing.

4.3.2.1 Scoring overview

	Weight Factor	NVIDIA Jetson Nano	NVIDIA Jetson TX2	NVIDIA Jetson Xavier NX
CUDA Cores	8	-1	1	2
Memory	7	1	2	2
Energy consumption	7	1	0	-1
Availability	6	2	2	-2
Cost	5	2	-2	-2
Weighted total score	-	28	24	1

Table 4.4: Handheld computing device scoring table.

Table 4.4 shows the scores of each hardware platform and their weighted total scores. It can be seen that the NVIDIA Jetson Nano has the highest weighted total score and is therefore the favorable single board computer development kit for this project. The scoring of the different options is motivated in the sections below.

4.3.2.2 CUDA cores scoring

The NVIDIA Jetson Nano has a score for *CUDA cores* of 0, while the NVIDIA Jetson TX2 has a score of 1 and the NVIDIA Jetson Xavier NX has a score of 2. This is due to the fact that the NVIDIA Jetson Nano contains the least amount of CUDA cores as shown in Table 4.3. The NVIDIA Jetson TX2 contains 256 CUDA cores, double the amount of CUDA cores on the NVIDIA Jetson Nano. The NVIDIA Jetson Xavier NX contains 384 CUDA cores, three times the amount of CUDA cores on the NVIDIA Jetson Nano.

4.3.2.3 Memory scoring

The NVIDIA Jetson Nano has a *memory* score of 1, while the NVIDIA Jetson TX2 and NVIDIA Jetson Xavier NX have a score of 2. This is due to the fact that with 4GB available RAM on the NVIDIA Jetson Nano, the computer is expected to be fully capable to execute the computations necessary for landmark measurement acquisition, resulting in a score of 1. The fact that the NVIDIA Jetson TX2 and the NVIDIA Jetson NX have 8GB of RAM, means that more memory can be utilized for additional functionality on the handheld measurement device, increasing flexibility. The *memory* of these two platforms therefore has a score of 2.

4.3.2.4 Energy consumption scoring

The NVIDIA Jetson Nano has an *energy consumption* score of 1, the NVIDIA Jetson TX2 has an *energy consumption* score of 0 and the NVIDIA Jetson Xavier NX has an *energy consumption* score of -1. As can be seen in Table 4.3, the NVIDIA Jetson Nano uses less power than the other two options, meaning that it consumes less energy. The NVIDIA Jetson Nano therefore has a score of 0. The NVIDIA Jetson Xavier NX consumes most energy and therefore has the lowest score of -1. The NVIDIA Jetson TX2 is right in between the other two options regarding power consumption and therefore receives a score of 0.

4.3.2.5 Availability scoring

Both the NVIDIA Jetson Nano and the NVIDIA Jetson Xavier NX receive an *availability* score of 2, while the NVIDIA Jetson TX2 receives an *availability* score of -2. The NVIDIA Jetson Nano and the NVIDIA Jetson Xavier NX are both readily available at online webstores with delivery within 2 days. The NVIDIA Jetson TX2 development board is out of stock and has a delivery time of over 2 weeks at multiple online stores in The Netherlands at the time of writing. As the delivery time is so much longer, this option receives an *availability* score of -2.

4.3.2.6 Cost scoring

The NVIDIA Jetson nano receives a *cost* score of 2, while the NVIDIA Jetson TX2 and the NVIDIA Jetson Xavier NX both have a *cost* score of -2. With the NVIDIA Jetson nano having a price of 105 euros, the cost is very low compared to the price of over 400 euros of the NVIDIA Jetson TX2 and the NVIDIA Jetson Xavier NX. The NVIDIA Jetson Nano therefore scores highest, while the other options score lowest.

4.3.3 The central computer GPU

	NVIDIA RTX 2080	NVIDIA RTX 2080 SU	NVIDIA RTX 2080 TI	NVIDIA RTX 3080
CUDA Cores	2944	3072	4352	8404
Tensor Cores	368	384	544	272
RAM (GB)	8	8	11	10
Price (€)	729	795	1150	918

Table 4.5: Computer hardware options for the desktop GPU.

The central computer GPU is the GPU that will be used for acceleration of the ES-EKF SLAM computations. The computer that will be used is a desktop computer in order to keep development effort low as the development computer is always accessible to the developer. The options for the desktop GPU are given in Table 4.5. This table contains a list of high-end CUDA enabled GPUs for desktop computers that carry both CUDA cores and Tensor cores.

4.3.3.1 Scoring overview

	Weight Factor	NVIDIA RTX 2080	NVIDIA RTX 2080 SU	NVIDIA RTX 2080 TI	NVIDIA RTX 3080
CUDA Cores	8	0	0	1	2
Tensor Cores	10	1	1	2	0
Memory	7	1	1	2	2
Availability	6	2	2	1	-2
Cost	5	1	1	-2	0
Weighted total score	-	34	34	38	18

Table 4.6: Desktop GPU scoring table.

Table 4.6 summarizes the scoring of each GPU option and provides their weighted total scores. It can be seen that the NVIDIA RTX 2080 TI has the highest weighted total score and is therefore selected to be used in the central computing system for the computation of the ES-EKF SLAM algorithm. The scoring of the different options is motivated in the sections below.

4.3.3.2 CUDA cores scoring

The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU have a *CUDA core* score of 0, the NVIDIA RTX 2080 TI has a *CUDA core* score of 1, while the NVIDIA RTX 3080 has a *CUDA core* score of 2. These scores are derived from Table 4.5, which shows that the NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU have the lowest amount of CUDA cores available, while the NVIDIA RTX 2080 TI has over 1000 CUDA cores more than the other two NVIDIA RTX 2080 options. The NVIDIA RTX 3080 has almost double the amount of CUDA cores compared to the NVIDIA RTX 2080 TI and therefore scores the highest.

4.3.3.3 Tensor cores scoring

The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU both have a *Tensor core* score of 1, while the NVIDIA RTX 2080 TI has a *Tensor core* score of 2 and the NVIDIA RTX 3080 has a *Tensor core* score of 0. As seen in Table 4.5, the NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU both provide a moderately high amount of Tensor cores. The NVIDIA RTX 2080 TI provides the most Tensor cores and therefore receives the highest score. The NVIDIA RTX 3080 provides only half the number of Tensor cores that the NVIDIA RTX 2080 TI provides and therefore receives the lowest score.

4.3.3.4 Memory scoring

The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU both have a *memory* core score of 1, while the NVIDIA RTX 2080 TI and the NVIDIA RTX 3080 have a *memory* score of 2. As seen in Table 4.5, the latter two GPUs provide over 2 GB of graphical RAM more than the NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU. The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU, however, already provide enough memory to provide the memory needed for ES-EKF SLAM computations with 1000 landmarks. These GPUs therefore receive a score of 1, while the GPUs with more memory receive a score of 2.

4.3.3.5 Availability scoring

The NVIDIA RTX 2080 and the NVIDIA RTX 2080 both have an *availability* score of 2, while the NVIDIA RTX 2080 TI has an *availability* score of 1 and the NVIDIA RTX 3080 has an *availability* score of -2. At the time of writing, the demand for graphics cards is high, raising the price of the GPUs and decreasing the availability. The NVIDIA RTX 3080 is the newest GPU in the list and is in very high demand, but as supply is low, delivery times are very uncertain and may be weeks. Therefore, the NVIDIA RTX 3080 has the lowest *availability* score of -2. The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU are on the other hand easily available at online stores with delivery within two days. This gives these GPUs an *availability* score of 2. Additionally, the NVIDIA RTX 2080 TI is also high in demand. However, delivery times for this GPU are up to a week. This GPU therefore has a score of 1.

4.3.3.6 Cost scoring

The NVIDIA RTX 2080 and the NVIDIA RTX 2080 SU both have a *cost* score of 1, while the NVIDIA RTX 2080 TI has a *cost* score of -2 and the NVIDIA RTX 3080 has a *cost* score of 0. It can be seen in Table 4.5 that the price of the NVIDIA RTX 2080 TI is the highest, giving it the lowest score for price of -2. The price of the NVIDIA RTX 2080 and the NVIDIA RTX 2080 is both below 800 euros, which is moderately less expensive than the other options. These options therefore have a *cost* score of 1. The NVIDIA RTX 3080 has a price between the NVIDIA RTX 2080 TI and the NVIDIA RTX 2080 SU, giving it a score of 0.

5 Implementation

This chapter discusses the system design and the development method used during the project and concludes with a description of the resulting system.

The design space exploration guided the decision-making process in order to develop a design of the system for ES-EKF SLAM for handheld perfusion imaging. As discussed in Section 3.4, this system is required to acquire landmark measurements with a rate of 10 measurements per second and process these measurements in real time, while providing a platform compatible with handheld perfusion imaging.

5.1 Proposed Implementation of Visual Error State EKF-SLAM for the HAPI

This section presents the proposed implementation of the Visual ES-EKF SLAM algorithm for the HAPI. The system architecture and the individual components of the system are discussed and a motivation is provided for the reason behind the design decisions.

As discussed in Chapter 4, the optimal computer architecture for the ES EKF-SLAM computations is the GPU with a manycore SIMT architecture and Tensor cores. This architecture allows to highly parallelize the different matrix and vector operations within the EKF-SLAM algorithm and accelerate the matrix-matrix multiplications. The GPU also allows to accelerate parallelizable algorithms, such as feature finding, descriptor extraction, descriptor matching, and landmark addition and removal. The optimization of the front-end computer vision algorithms is also often proposed by the works discussed in Section 1.1. This makes the use of the GPU applicable in both the handheld device and the desktop computer, as the decision was made to perform the landmark measurement acquisition on the handheld device and the ES-EKF SLAM computations on the desktop computer with GPU.

Using the CUDA programming model offers the advantage of scalability: as the number of streaming multiprocessors increases with newer GPUs, the CUDA GPU automatically takes advantage of the increased amount of processors, further increasing parallelism. This way, the accelerated ES-EKF SLAM implementation can potentially take advantage of CUDA compatible GPUs with more SIMT cores and more device memory in the future.

5.1.1 System architecture

In order to increase parallelism within the system, the landmark measurement acquisition and ES-EKF SLAM computations are performed simultaneously. This means that while the landmark measurements acquired at time t are being used to update the state of the system, the handheld device is already acquiring the landmark measurements at time $t + 1$, as shown in Figure 5.1.

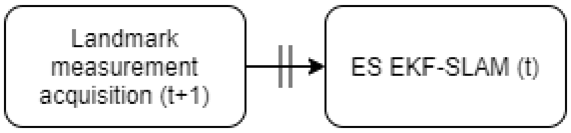


Figure 5.1 Diagram showing the parallelization between the landmark measurement acquisition and the ES-EKF SLAM computation.

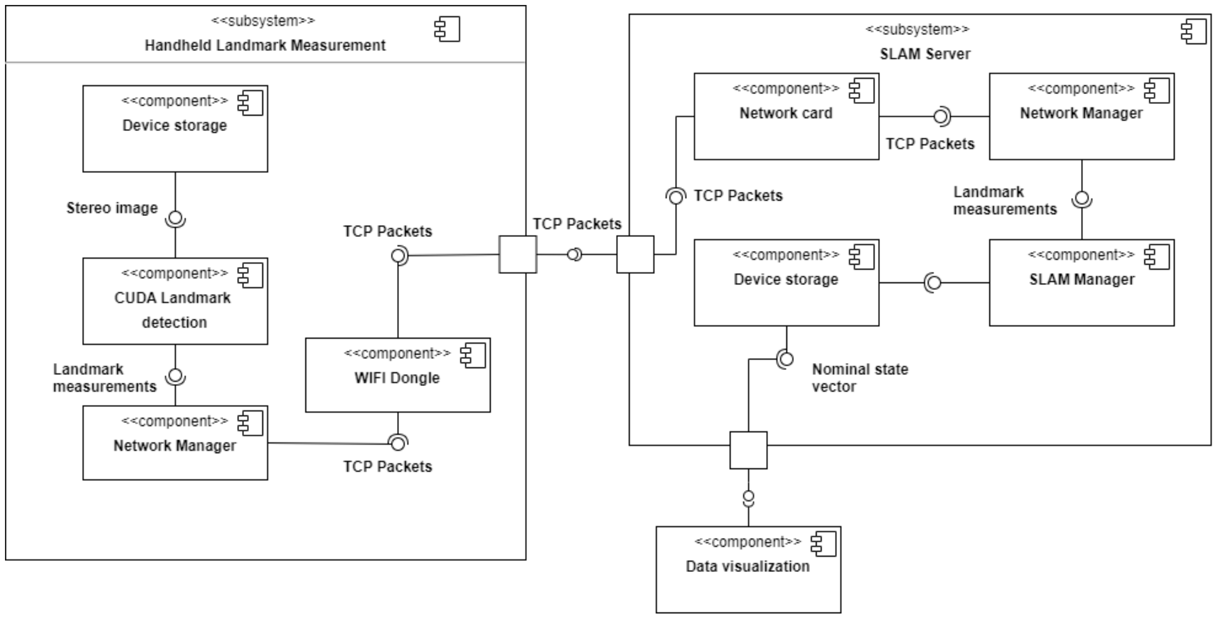


Figure 5.2: Component diagram of the proposed system.

Figure 5.2 shows the system architecture as a component diagram. The landmark measurement acquisition is performed on a NVIDIA Jetson Nano, and the landmark measurements are transmitted over Wi-Fi using a USB Wi-Fi adapter. The measurements are then received by a desktop computer, the SLAM Server, which is connected to a Wi-Fi router by a LAN Cable. The SLAM Server processes the landmark measurements and uses them for the ES-EKF SLAM computations. Transmission over Wi-Fi to a local network was chosen as this network infrastructure is readily available and is relatively simple to implement using proper libraries, such as the Boost.Asio library (Kohlhoff, 2021).

5.1.2 Handheld Device

The proposed handheld device consists of a NVIDIA Jetson Nano Development Kit, which contains the NVIDIA Jetson Nano single board computer and offers IO ports to interface two cameras through two CSI ports for image acquisition and multiple USB ports to connect the Wi-Fi adapter. The NVIDIA Jetson Nano contains a 128-core Maxwell GPU, a Quad-core ARM A57 CPU clocked at 1.43 GHz, and 4GB LPDDR4 memory.

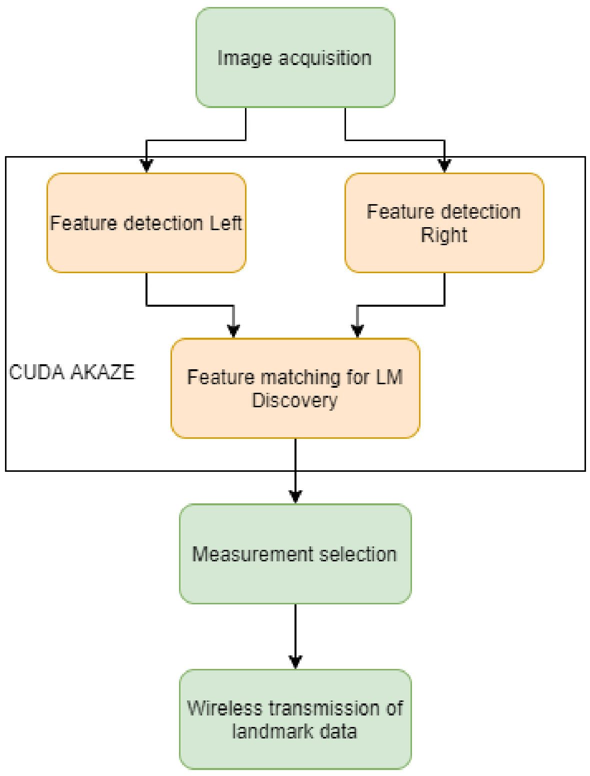


Figure 5.3 Dataflow of the landmark measurement acquisition.

The NVIDIA Jetson Nano runs the landmark measurement acquisition software, of which a data flow diagram is shown in Figure 5.3. The diagram shows which computing device is used for certain computations and that the CUDA AKAZE library is proposed to be used for the detection, extraction and matching of KAZE features.

This software acquires two image frames from a stereo camera every 0.1 seconds. Each image then goes through a feature detection algorithm which finds KAZE features in the images as described by Alcantarilla, Bartoli, & Davidson (2012). The descriptor extraction algorithm then collects the descriptor for each feature for feature matching. The feature finding, descriptor extraction and matching are performed using the CUDA AKAZE library, which offers a parallelized way of performing the operations using a CUDA compatible GPU. The feature finding and descriptor extraction for each image is independent of each other and can therefore run in parallel.

After the features are matched, the strongest matches are selected in order to filter out weak and wrong matches and to decrease the maximum number of visible landmarks L. This also decreases the problem size, as discussed in Chapter 2.3. The selected measurements are then collected, listing their pixel location in both images, left feature descriptor for landmark matching, strength, and color. These measurements, together with the stereo camera calibration data, are then ready for transmission over Wi-Fi using the Boost.Asio library so that they can be used to update the state in the ES-EKF SLAM software.

In order to start the execution of the landmark measurement acquisition with different settings, a command line interface is implemented. Through this command line interface different accelerations can be enabled or disabled and the input size of the stereo images can be set.

5.1.3 SLAM Server

The SLAM Server is a desktop computer which runs the ES-EKF SLAM Server software. The computer contains an Intel Core I7 6700K clocked at 4.00 GHz, a Gigabyte RTX 2080 TI, and 16 GB of DDR4 RAM clocked at 2400 MHz. The Gigabyte RTX 2080 TI contains 4352 CUDA Cores, 544 Tensor Cores, and 11 GB of DDR6 memory.

The SLAM Server runs the ES-EKF SLAM software, which is described below. The dataflow diagram of the software is shown in Figure 5.4. The software is based on the ES EKF-SLAM implementation developed by Shah and Van der Heijden (Shah, 2020) in MATLAB and is written in C++.

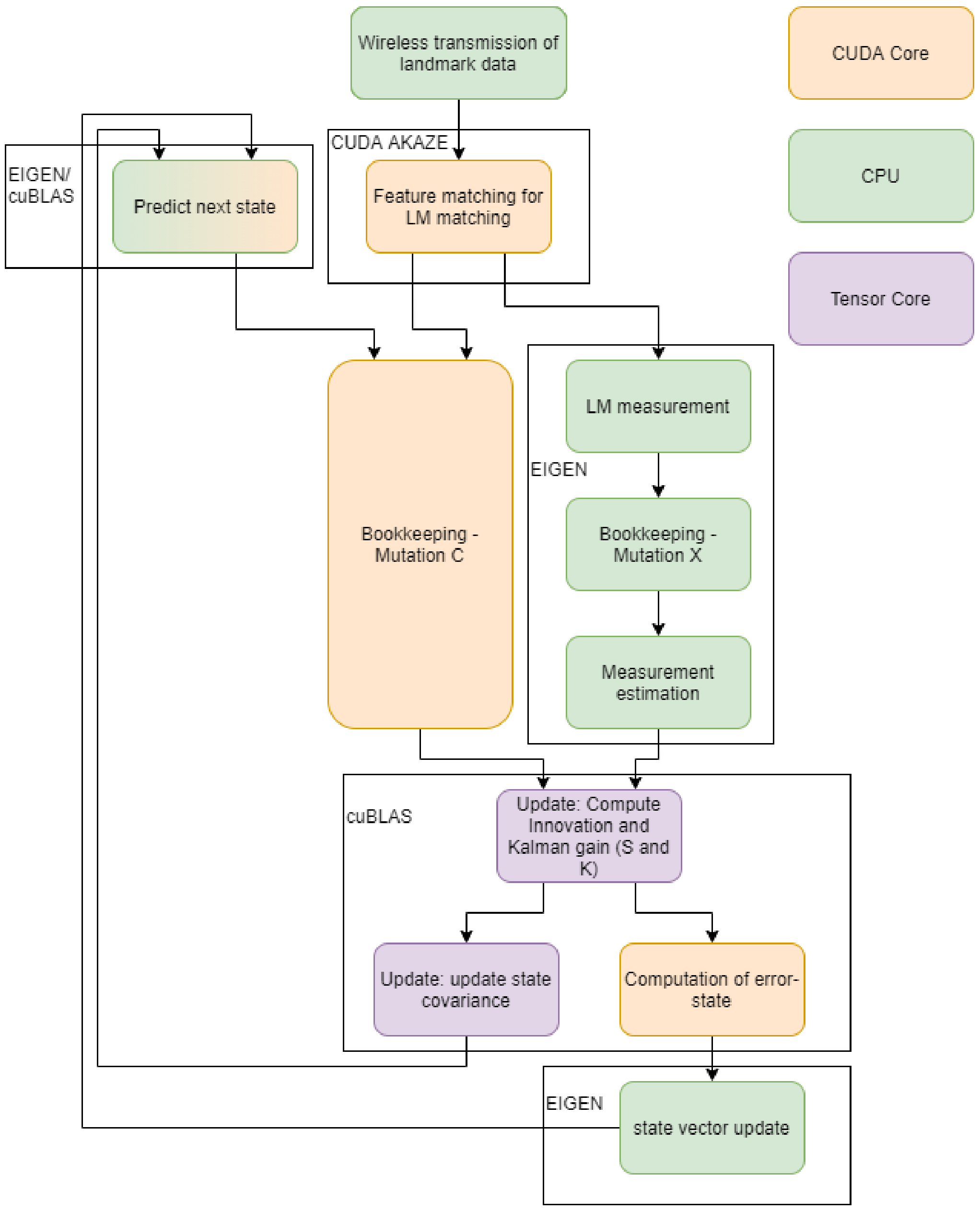


Figure 5.4 Data flow diagram of the SLAM Server software for ES-EKF SLAM.

The SLAM Server waits to receive landmark measurements from the handheld device, acting as a server for the device to connect to. Incoming measurements are stored and processed by the ES-EKF SLAM algorithm. As a set of landmark measurements is used for the ES EKF-SLAM computation, they first enter the landmark association algorithm where landmark measurements are tied to landmarks that are already being tracked. Duplicated landmarks are detected and removed. The landmark association algorithm uses the same parallelized descriptor matcher provided by the CUDA

AKAZE library that the handheld device uses. This results in a set of visible landmarks which consists of tracked landmarks with a measurement at that timestep, and measurements to new landmarks.

These sets then enter the *bookkeeping* algorithm where landmarks are deleted from the landmark pool in case of landmark duplicates or if space for new landmarks needs to be freed. In such case, the landmarks are removed from the nominal state vector and the covariance matrix. In order to minimize data transmission between the host memory and device memory, the state covariance matrix is stored in device memory. This also enables for parallelization of the landmark removal algorithm using a custom CUDA kernel to increase the speed of this algorithm. The same holds for the addition of new landmarks to the nominal state vector and covariance matrix. Operations on the nominal state vector are performed using the Eigen Library for linear algebra in order to also utilize the CPU while the GPU is performing operations on the covariance matrix. Before landmarks are added to the state, the strongest landmarks are selected based on their strength. The position relative to the camera of these landmarks is estimated in order to build a measurement vector. The next step before the ES-EKF *update* is the estimation of measurements and computing matrix H, which is also performed on the CPU using the Eigen library.

Once the *bookkeeping* algorithm is finished and the measurement estimation is done, the resulting measurement error and matrices H and Cv are transferred to the GPU device memory for the ES-EKF *update*. This *update* step is performed using the cuBLAS library. Unfortunately, this library does currently not support use of the Tensor cores for FP64 operations. This, however, can be overcome by using the library written by Mukunoki, Ozaki, Ogita & Imamura (2020), which uses FP16 Tensor cores for FP64 matrix-matrix multiplication.

In order to reduce the problem size, it can be decided to limit the number of visible landmarks l by limiting the number of old landmarks that are used to update the state.

Once the error state is computed, it is transferred to the host memory such that it can be used to update the nominal state vector. The nominal state vector can then be stored and used for its application such as camera device tracking for handheld perfusion imaging.

The *prediction* step of the ES-EKF computation makes use of a combination of cuBLAS, as it contains a matrix-matrix multiplication containing the covariance matrix, and the Eigen library to predict the pose at the next timestep.

In order to start the execution of the ES-EKF SLAM computations with different settings, a command line interface is implemented. Through this command line interface different accelerations can be enabled or disabled and the different options regarding landmark management and pool size can be changed.

5.2 Development method

This section discusses the used development method, which consists of multiple iterations of implementation of individual accelerations of the algorithm. As the time for implementation is limited due to hardware shortages and considering the complexity of the ES-EKF SLAM algorithm, it is important to follow a streamlined and disciplined development method where the accelerations are implemented in order of expected increase in performance, starting from a functional prototype.

5.2.1 The implementation cycle

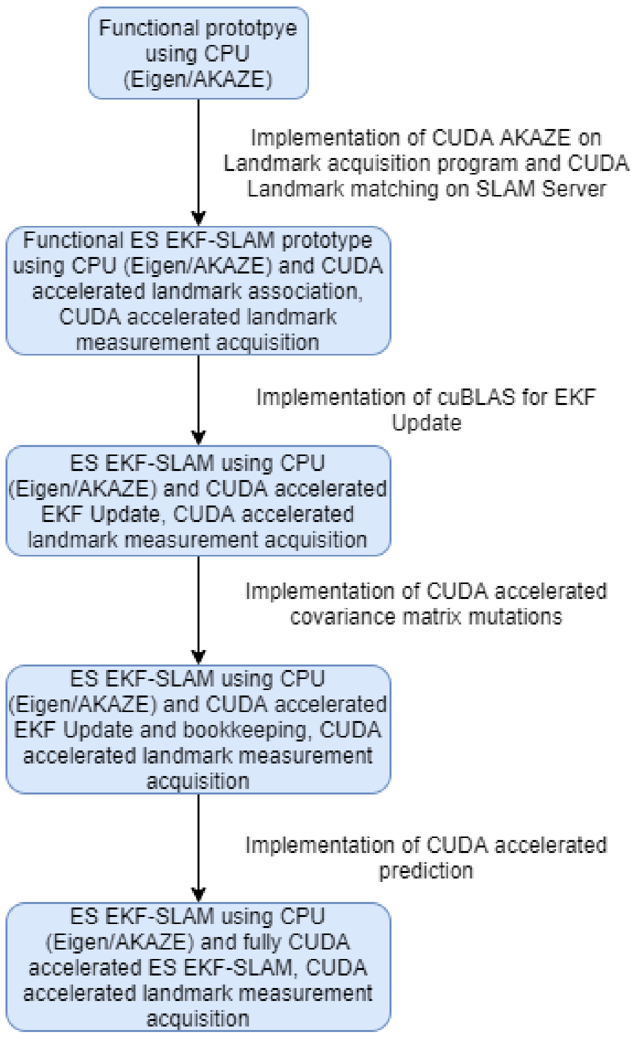

Functional prototpye using CPU (Eigen/AKAZE)
Implementation of CUDA AKAZE on Landmark acquisition program and CUDA Landmark matching on SLAM Server
Functional ES EKF-SLAM prototype using CPU (Eigen/AKAZE) and CUDA accelerated landmark association, CUDA accelerated landmark measurement acquisition
Implementation of cuBLAS for EKF Update
ES EKF-SLAM using CPU (Eigen/AKAZE) and CUDA accelerated EKF Update, CUDA accelerated landmark measurement acquisition
Implementation of CUDA accelerated covariance matrix mutations
ES EKF-SLAM using CPU (Eigen/AKAZE) and CUDA accelerated EKF Update and bookkeeping, CUDA accelerated landmark measurement acquisition
Implementation of CUDA accelerated prediction
ES EKF-SLAM using CPU (Eigen/AKAZE) and fully CUDA accelerated ES EKF-SLAM, CUDA accelerated landmark measurement acquisition

Figure 5.5 Overview of Iterations of the system during development.

During the implementation, a cycle was followed consisting of 4 steps: design, implementation, testing, and analysis. Each iteration of the cycle implements an acceleration based on its expected increase in performance. During the design step, the proposed system design and acceleration methods are combined into a design of the software implementation. This software design was then implemented and functionally tested such that it is equal to its counterpart written in MATLAB. The implementation was then profiled in order to find bottlenecks, which is where the next major increase in performance is expected to be. This resulted in multiple iterations of the system, as shown in Figure 5.5.

In the following sections each step of the cycle is explained in more detail.

5.2.1.1 Step 1: design

During each design step, the acceleration with the highest expected speedup was taken as an input. The implementation of the acceleration was then designed based on the proposed system design and the functional requirements of the function or step in the ES-EKF SLAM algorithm that is accelerated. It was then decided which optimization methods can be applied in order to maximize the acceleration of the function.

5.2.1.2 Step 2: implementation

Once it is clear from the design step how a software acceleration is supposed to be implemented, the software design is implemented on the system. The old implementation of the functionality is kept, while a second version of the functionality is developed which includes the chosen optimization methods. This results in two implementations which can be used to provide a comparison of the execution time, measuring the speedup of the accelerated implementation.

5.2.1.3 Step 3: testing

Testing was performed after each implementation. During the testing step, the implemented acceleration is debugged and functionally verified to be equal to its non-accelerated counterpart.

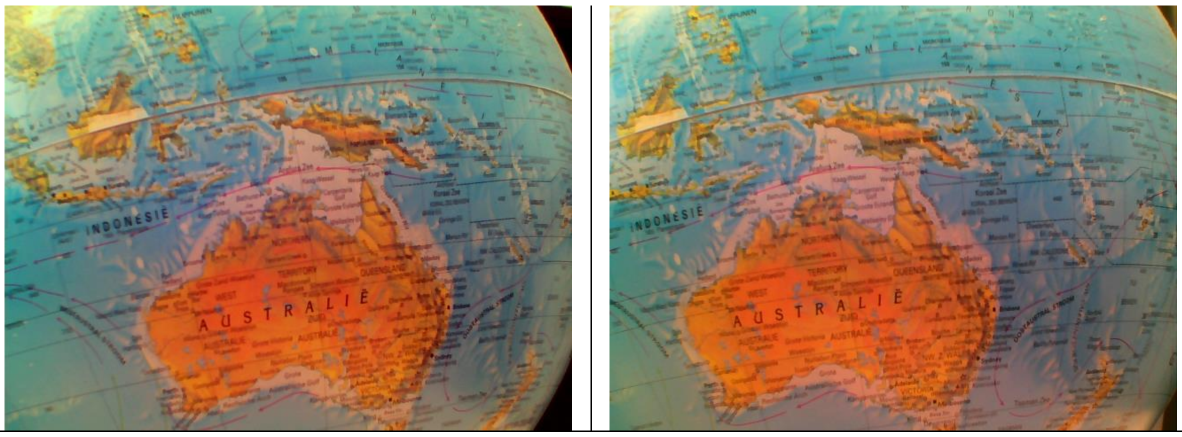


Figure 5.6 Stereo image example. **A:** Left image example of the Globe dataset. **B:** Right image example of the Globe dataset.

For testing, a constant input was used for landmark measurement acquisition. This input is the Globe dataset, of which a sample stereo image is shown in Figure 5.6. This dataset contains a set of stereo images, imaging the surface of a globe. These images were acquired by keeping the camera stationary at a static distance from the globe, while the globe is manually rotated, creating the illusion of the camera rotating around the globe. This way, the movement of the camera is relatively constant and provides a good reference, as it is supposed to be circular. Additionally, the round shape of the globe provides a good reference for the expected position of the measured landmarks, as these landmarks are expected to be on a spherical surface.

5.2.1.4 Step 4: analysis

During the analysis step, the software implementation was profiled in order to get an insight of the performance of the software implementation and which part of the algorithm can be further accelerated. The results from this step were then used during the design step of the next iteration, where the acceleration with the highest potential speedup is selected.

5.3 Implementation result

This section presents the resulting accelerated ES-EKF SLAM implementation, how the different optimizations are implemented, and the qualitative performance of this implementation compared to its MATLAB counterpart.

The implemented system is almost fully implemented as proposed in Section 5.1. Unfortunately, as Figure 5.5 shows, not all optimizations proposed in the original system design could be implemented. Parallel execution of the feature finding and description extraction algorithms for both images in the landmark measurement acquisition software, parallel execution of the ES-EKF *prediction* and landmark association, as well as the execution of matrix-matrix multiplications on Tensor cores could not be implemented during the project.

The resulting system has an architecture as shown in the component diagram in Figure 5.2 and uses the hardware discussed in Section 5.1 to run the landmark measurement acquisition software the SLAM Server software. Each software implementation is discussed further in the coming sections.

5.3.1 Landmark measurement acquisition implementation

As discussed in Section 5.1.2, the landmark measurement acquisition software performs the image processing steps to acquire landmark measurements in the form of 2D image locations of the landmarks and their descriptors. No cameras were implemented for image acquisition, but images were acquired from the device storage. The acquired landmark measurements were transmitted to the SLAM Server over Wi-Fi using a USB Wi-Fi adapter.

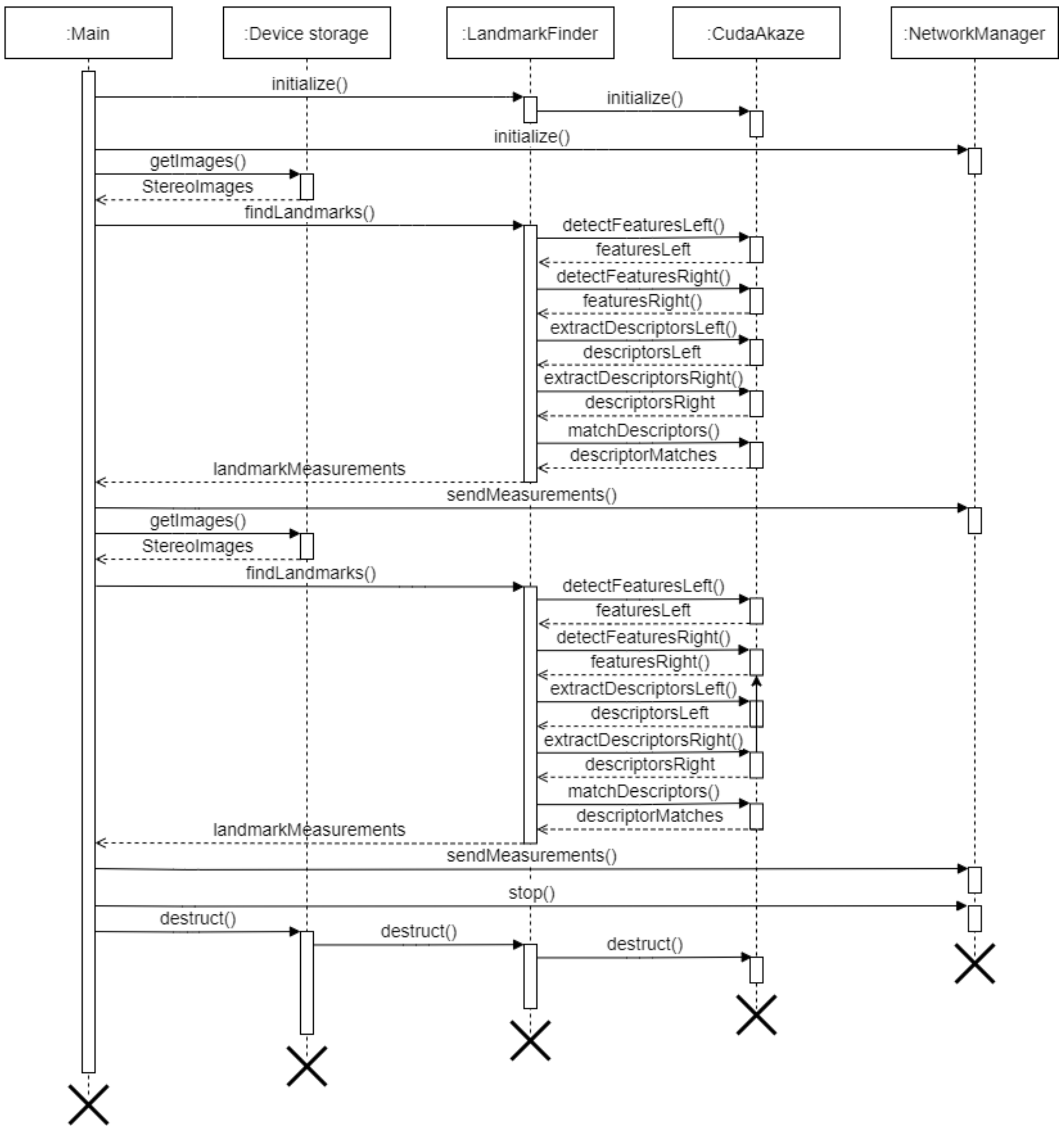


Figure 5.7: Sequence diagram showing the functioning of the landmark measurement acquisition software.

Figure 5.7 shows a sequence diagram that describes how the landmark measurement acquisition software is implemented. The diagram shows two iterations of landmark measurement acquisition and the termination of the measurement. As can be seen, a landmark measurement is started by retrieving a stereo image pair from the device storage. The stereo images are then compensated for lens distortion and the resulting undistorted images are then given to the *LandmarkFinder* for landmark finding, matching, and filtering. During this process, the CUDA AKAZE library is used for the parallelized execution of the computer vision algorithms on the GPU of the NVIDIA Jetson Nano. The matched features are then filtered in order to select the 200 strongest matches, which are used as landmark measurements. The landmark measurements are then returned in order for them to be passed to the *NetworkManager*, which asynchronously sends the landmark measurements to the SLAM Server using the Boost.Asio library.

The implemented landmark measurement acquisition software contains a camera calibrator, which can be used to calibrate the stereo camera to obtain the camera parameters necessary for landmark depth estimation. This feature is used as a support of the ES EKF-SLAM system.

The command line interface provides the following options shown in Table 5.1 on startup of the software.

Command	Option
-M	Perform landmark measurement acquisition
-C	Perform stereo camera calibration
-f	Filename of camera calibration file (input for measurement, output for calibration)
-p	Path to input images
-l	Filename of left input images
-r	Filename of right input images
-e	Filename extension of input images
-n	Number of images used for stereo calibration
-o	Output filename of landmark measurements
-m	Desired descriptor matcher: {ACPU, AGPU}
-ic	Width of the input images
-ir	Height of the input images
-a	Desired output filename of the runtime analysis

Table 5.1: Command line interface options of the landmark measurement acquisition software.

5.3.2 SLAM Server implementation

The SLAM Server receives incoming landmark measurements and uses them to perform the ES-EKF SLAM computations, which are acclerated using the GPU. The software runs on two threads: one thread executes the receiving of incoming landmark measurements, and a second thread performs the ES-EKF SLAM computations.

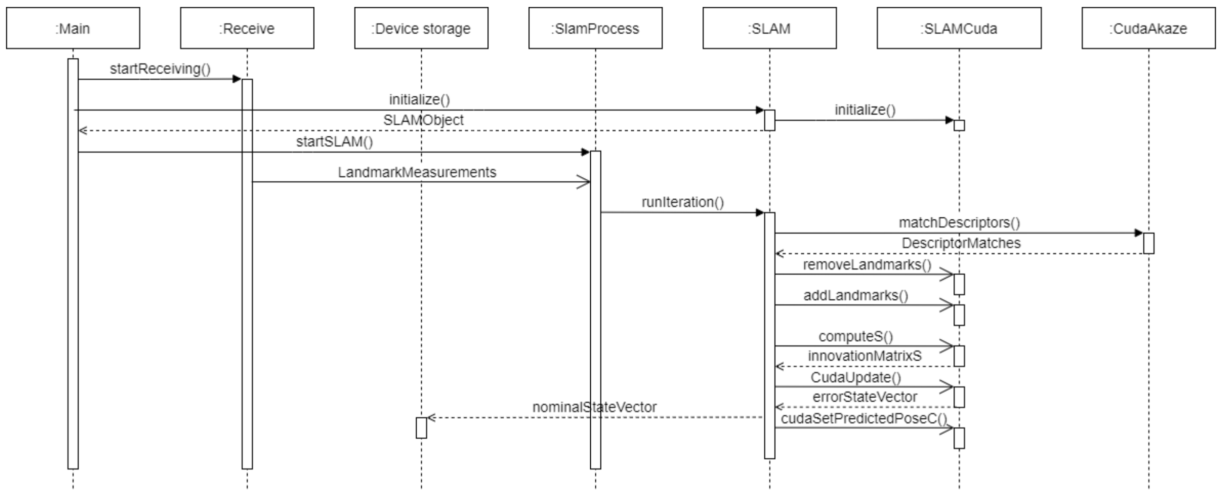


Figure 5.8: Sequence diagram showing the functioning of the SLAM Server software.

Figure 5.8 shows the sequence diagram representing the structure of the SLAM Server software implementation. The diagram shows a single iteration of the ES-EKF SLAM algorithm after the software is started. When the software starts, the two threads for landmark measurement receival and ES-EKF SLAM computation are started. During this bootup process, the SLAM object initializes the CUDA execution of the ES-EKF SLAM computations by reserving the memory needed for the different vectors and matrices used in the ES-EKF computations. As discussed in Section 5.1.3, the error state covariance matrix is stored in this memory in order to decrease data transmission between the host processor and the GPU device.

Once landmark measurements arrive, the landmark measurements are asynchronously passed to the *SlamProcess*, which passes the measurements to the SLAM object which performs the execution of the ES-EKF SLAM algorithm. During the *bookkeeping* step, the CUDA AKAZE library is used to accelerate the landmark descriptor matching and a CUDA implementation of the landmark removal and addition is used to parallelize the error state covariance matrix mutations. In order to further accelerate the removal of landmarks from the covariance state vector, landmarks that are added later have a priority to be removed, as removing these landmarks requires less rows and columns to be removed in the state covariance matrix. Then the cuBLAS library is used to perform the most time expensive computations of the error state extended Kalman filter for SLAM. As discussed in Section 5.1.3, operations on the nominal state vector are performed using the Eigen Library for linear algebra in order to also utilize the CPU while the GPU is performing operations on the covariance matrix. The Eigen library is also used to invert the innovation matrix S for the computation of the Kalman Gain matrix G, as this could not be implemented on the GPU in the given timeframe. Once the state vector is updated, the vector is stored on the computer storage so that it can analyzed and visualized using MATLAB.

The command line interface of the SLAM Server provides the following options shown in Table 5.2 on startup of the software.

Command	Option
- CUDA	Perform GPU accelerated ES EKF-SLAM.
- EIGEN	Perform non-accelerated ES EKF-SLAM.
- m	Desired descriptor matcher: {ACPU, AGPU, MLB}.
-s	Desired option for saving either the nominal state vector or all data used for debugging: {STATE, ALL}.
-k	Maximum number of landmarks in the pool.
-n	Maximum number of new landmarks at every iteration.
-o	Maximum number of old landmarks used for the ES EKF state update.
-p	Percentage of added landmarks when the pool is full.

Table 5.2: Command line interface options of the SLAM Server software.

5.3.3 Qualitative results of the ES-EKF SLAM execution

This section provides a comparison between the outputs of the accelerated ES-EKF SLAM implementation and the MATLAB implementation. The output of the accelerated ES-EKF SLAM algorithm differs from the MATLAB implementation. This is because the qualitative performance of the feature finding implementation differs from the MATLAB implementation and, as explained in Section 5.3.2, changes have been made to the *bookkeeping* algorithm to further increase the acceleration of the algorithm.

In order to evaluate the qualitative performance of the accelerated implementation, the resulting camera pose and velocity over time from each implementation is compared. This is done by computing the Root Mean Square Error (RMSE) value of the camera pose and the camera velocity of the accelerated implementation, which are the first 13 values of the nominal state vector.

The results are gathered by executing the algorithms with the following settings:

- Maximum number of landmarks in the pool: 8000
- Maximum number of visible landmarks: 200
- Maximum number of landmarks added at each iteration: 20
- Percentage of new landmarks added when the pool is full: 50%
- Maximum number of old landmarks used for the ES-EKF *update* step: 200

Value	RMSE (mm)
p_x^w	117.242
p_y^w	27.143
p_z^w	89.033
q_0^w	0.079
q_1^w	0.028
q_2^w	0.096
q_3^w	0.035
$v_{wc,\mathrm{x}}^c$	23.222
$v_{wc,\mathrm{y}}^c$	9.230
$v_{wc,\mathrm{z}}^c$	5.995
$\omega_{wc,\mathrm{x}}^c$	0.021
$\omega_{wc,\mathrm{y}}^c$	0.053
$\omega_{wc,\mathrm{z}}^c$	0.009

Table 5.3: RMSE values of the pose estimation resulting from the accelerated ES-EKF SLAM implementation.

Table 5.3 gives an overview of the RMSE values for each pose variable. It can be seen that the RMSE values of the orientation and the angular velocity are relatively low, while there is a large error in the estimation of the position and the translational velocity, mainly in the x-direction. This error is also visible when comparing the resulting landmark maps.

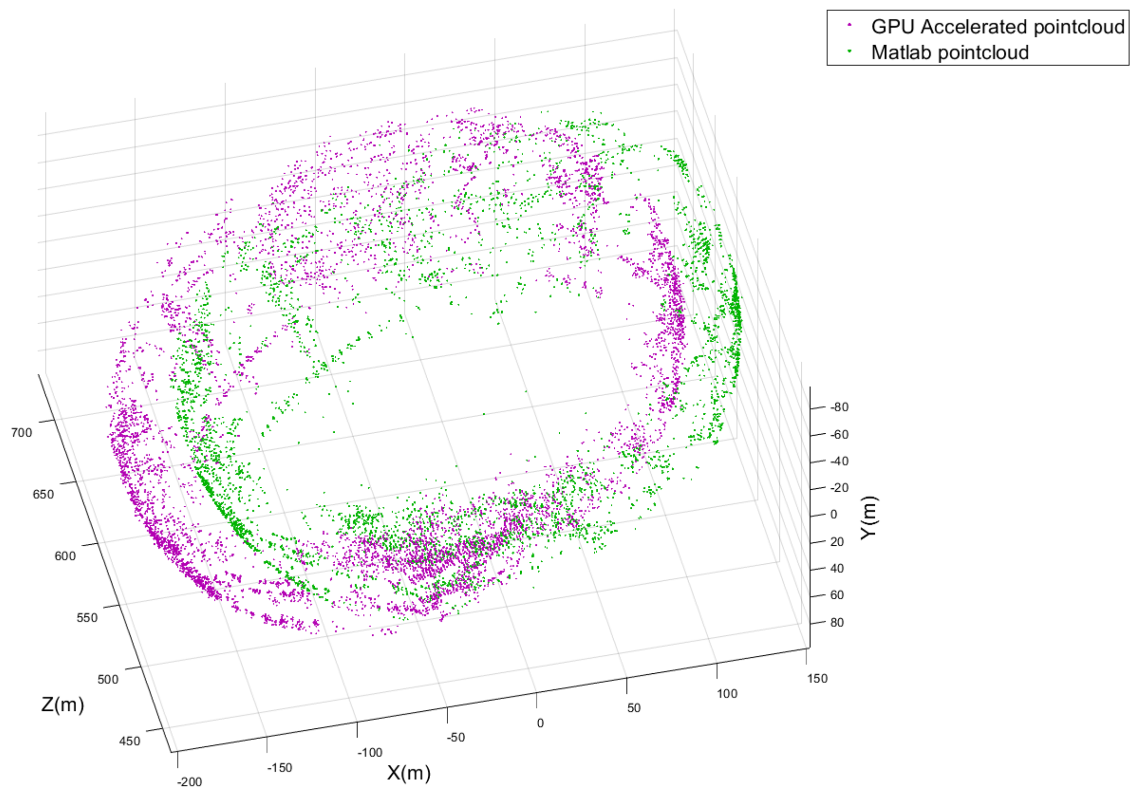


Figure 5.9: Comparison of the resulting landmark maps of the accelerated implementation and the MATLAB implementation of the ES-EKF SLAM algorithm.

Figure 5.9 provides a comparison of the landmark maps resulting from the execution of the ES-EKF SLAM algorithm by the accelerated implementation and by the MATLAB implementation. It can be seen that the landmarks resulting from the accelerated implementation are shifted and rotated with respect to the landmarks resulting from the MATLAB implementation.

It can thus be concluded that the qualitative performance of the accelerated algorithm differs from the MATLAB implementation. There is a substantial difference in the estimated camera pose and velocity and the landmark map of the accelerated implementation is shifted and rotated with respect to its MATLAB counterpart.

6 Evaluation

The evaluation of the system is divided in two parts: a one-by-one comparison of the system performance against its MATLAB counterpart with equal input, and a worst-case execution time evaluation, which is used to determine the real-time performance of the system with different settings.

The one-by-one comparison assesses the increase in performance of the implemented system for Error State EKF-SLAM. The runtime performance of the system is evaluated and compared to the MATLAB implementation and, if possible, to a CPU implementation without GPU accelerations. This will provide information about the overall performance increase of the designed system with respect to the MATLAB prototype.

The worst-case execution time evaluation will provide information about the worst-case execution times of the landmark measurement acquisition and the ES-EKF SLAM algorithm. The execution times of these algorithms are highly dependent on the input of the algorithm, which is the size of the input image for landmark measurement acquisition, and the size of the state vector for the ES-EKF SLAM algorithm. This analysis provides a basis for choosing a size of the input image and the maximum size of the state vector in order to ensure that the system measures and analyses 10 measurements per second in real time.

6.1 Evaluation method

The Globe dataset is used as input for landmark measurement acquisition in order to keep the input consistent. A maximum number of landmarks in the pool is set to 8000. This is slightly below the maximum number of landmarks possible on the RTX 2080 TI, which is 8200. The maximum number of visible landmarks is set at 200, which is equal to the number of measurements per timestep, and the maximum increase in pool size is set at 20 landmarks. These settings help measuring the effect of the size of the state vector on the execution time of the ES-EKF SLAM algorithm as the size gradually increases.

To summarize, the results are gathered by executing the algorithms with the following settings:

- Maximum number of landmarks in the pool: 8000
- Maximum number of visible landmarks: 200
- Maximum number of landmarks added at each iteration: 20
- Percentage of new landmarks added when the pool is full: 50%
- Maximum number of old landmarks used for the ES-EKF *update* step: 200

Execution time is measured using inline software timers and the results are evaluated using MATLAB and is used to compute the speedup of an accelerated element and the speedup of the system as a whole. The speedup is computed as follows:

$$speedup = \frac{Execution\ time\ without\ acceleration}{Execution\ time\ with\ acceleration}$$

6.1.1 Evaluated landmark measurement acquisition elements

The evaluation of the landmark measurement acquisition is divided in three parts. The first part provides an analysis of the execution times of the feature finding and descriptor extraction algorithm with and without GPU acceleration. The second part consists of the runtime performance evaluation of the matching algorithm implemented on the GPU, CPU, and the MATLAB implementation. The third part provides an overview of the execution time of the landmark measurement acquisition, its speedup, and the time it takes for measurements to be transmitted over Wi-Fi compared to ethernet. The Wi-Fi adapter used is a TP-Link TL-WN823N USB Wi-Fi adapter. The size of the input images is 640 by 480 pixels for each image from the stereo camera.

The worst-case execution time evaluation of the landmark measurement acquisition consists of the evaluation of the effect of the input image size on measurement acquisition runtime performance. During this part, the runtime performance of the fully GPU accelerated landmark measurement acquisition algorithm on the handheld device is measured for different image sizes. This provides information about the maximum image size which supports real-time measurement acquisition.

6.1.2 Evaluated ES-EKF SLAM elements

The runtime performance of different parts of the ES-EKF SLAM implementation on the SLAM Server are independently measured in order to compare the impact of the implemented optimizations compared to the MATLAB implementation of the algorithm. These parts are the landmark matching implementation, the overall *bookkeeping* algorithm and the state landmark deletion and landmark addition algorithms, the overall ES-EKF implementation, consisting of the *update* step and the *predict* step. Finally, a comparison is performed between the fully optimized ES-EKF SLAM implementation, a CPU implementation without GPU acceleration, and the MATLAB implementation.

For the runtime performance analysis of the landmark matching implementation, three landmark matching implementations are evaluated: AKAZE GPU matching, AKAZE CPU matching, and the MATLAB descriptor matcher. Each matcher is fed the same set of measurements and compares the measurements against the landmarks in the pool.

For the runtime performance evaluation of the *bookkeeping* algorithm, the runtime performance of the complete *bookkeeping* algorithm is measured. Meanwhile, the runtime performance of the landmark addition and landmark deletion algorithms are individually measured in order to analyze their impact on the runtime performance of the *bookkeeping* algorithm. The runtime performance of the fully GPU accelerated implementation as well as the runtime performance of the CPU implementation without GPU acceleration is measured and compared to the runtime performance of the MATLAB algorithm. As the landmark matching algorithms are different, resulting in different matching results, the results of the MATLAB landmark matcher are used for all runtime performance measurements in order to have a consistent input in the *bookkeeping* algorithm.

The runtime performance measurement of the ES-EKF algorithm is performed by measuring the *update* and *predict* steps individually to analyze their influence on the runtime performance of the overall algorithm. Similar to the measurement of the runtime performance of the *bookkeeping* algorithm, the runtime performances of the fully GPU accelerated implementation, the CPU implementation without GPU accelerations, and the MATLAB implementation are measured. In order to keep the input to these algorithms consistent for these measurements, the same landmark association results are used as for the *bookkeeping* evaluation.

The number of old landmarks used for the state *update* is not limited for this evaluation, in order to measure the influence of the amount of visible landmarks on the execution time of the state *update* computation.

Finally, the overall runtime performance of the implementation is measured and compared to its MATLAB counterpart. During this evaluation, the same landmark measurements are used for all implementations. The runtime performance of the following implementations is measured:

- The MATLAB implementation.
- The SLAM Server CPU implementation without GPU accelerations.
- The fully GPU accelerated SLAM Server.

The worst-case execution time analysis of the ES-EKF SLAM algorithm consists of a comparison of the execution times of the algorithm for different maximum pool sizes K, while l is limited by limiting the amount of old visible landmarks to 30. This provides information on how the maximum pool size influences the worst-case execution time in order to decide what the maximum value can be for the algorithm to process landmark measurements to perform the SLAM loop in real-time following the real-time constraint set in Section 3.4.

6.2 Evaluation results

This section presents the results of the runtime performance evaluation discussed in the previous section.

6.2.1 Landmark acquisition

As discussed, the parts of the landmark acquisition discussed during the runtime performance analysis are the different descriptor matching implementations, the feature matching and descriptor extraction as implemented on the handheld device with GPU acceleration, compared to the MATLAB implementation, the overall measurement runtime, and the transmission time of the landmark measurements over Wi-Fi and ethernet.

6.2.1.1 Descriptor matching

Figure 6.1 shows the runtime performance of the different keypoint matching algorithm implementations. Figure 6.1 A shows the matching time at each timestep of the different feature matchers. It can be seen that the matcher executed on the CPU of the NVIDIA Jetson Nano is significantly slower than the MATLAB implementation running on the desktop computer and the matcher executed on the GPU of the NVIDIA Jetson Nano. As shown in Figure 6.1 B, executing the matching on the GPU of the handheld device results in a significant performance increase, which even outperforms the MATLAB matcher on the desktop computer. It can also be seen that the matching time varies with the amount of found features, which has mainly a significant impact on the feature matching time when performed on the CPU of the NVIDIA Jetson.

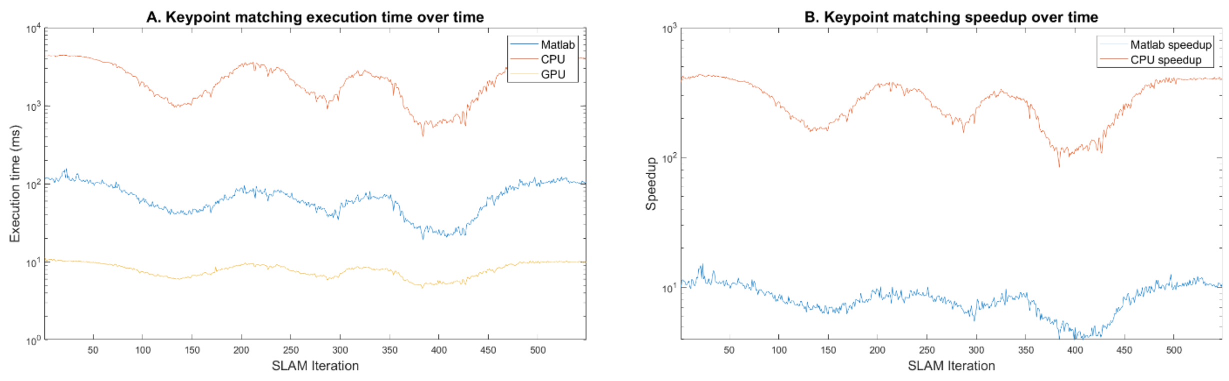


Figure 6.1 A. Landmark matching execution time of the MATLAB, CPU and GPU matchers over time. **B.** Speedup of the accelerated matcher in relation to the MATLAB and CPU matchers.

6.2.1.2 Feature finding and descriptor extraction

Figure 6.2 shows the runtime performance of the feature finding and descriptor extraction algorithms combined. The execution times include the processing of both images of the stereo camera. Figure 6.2 A shows the execution time of these algorithms implemented on MATLAB and with GPU acceleration over time. Figure 6.2 B shows the speedup of the GPU accelerated feature finding and descriptor extraction algorithm compared to the MATLAB implementation.

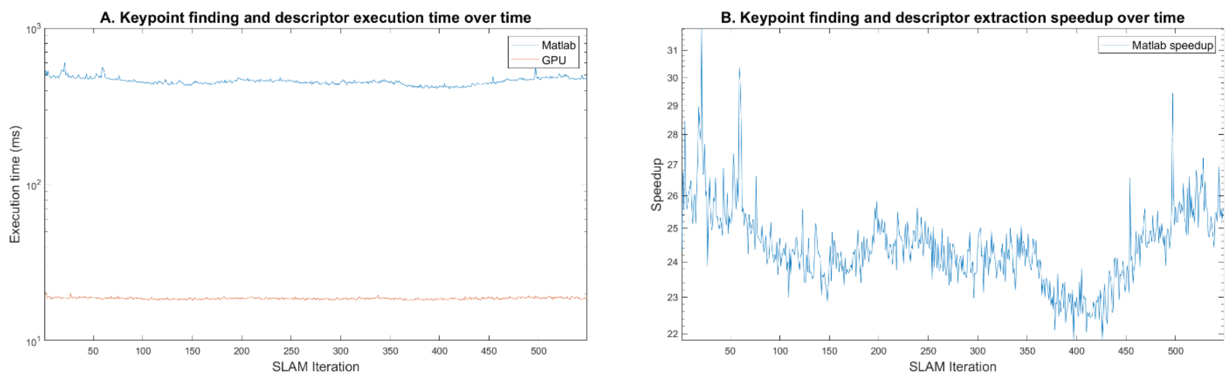


Figure 6.2 A. Execution time of the of the keypoint finding and descriptor extraction algorithms. **B.** Speedup of the GPU accelerated feature finding and descriptor extraction algorithm compared to the MATLAB implementation.

6.2.1.3 Full landmark acquisition

Figure 6.3 A shows the execution time of the full landmark measurement acquisition for the different implementations over time. Figure 6.3 B shows a comparison of the transmission time of the landmark measurements over Wi-Fi and over ethernet. Figure 6.3 C shows the speedup of the GPU accelerated implementation with respect to the MATLAB implementation and the non-GPU accelerated implementation. Finally, Figure 6.3 D shows the composition of the average landmark measurement acquisition execution time. This figure shows the average execution time of the different parts of the accelerated landmark measurement acquisition algorithm.

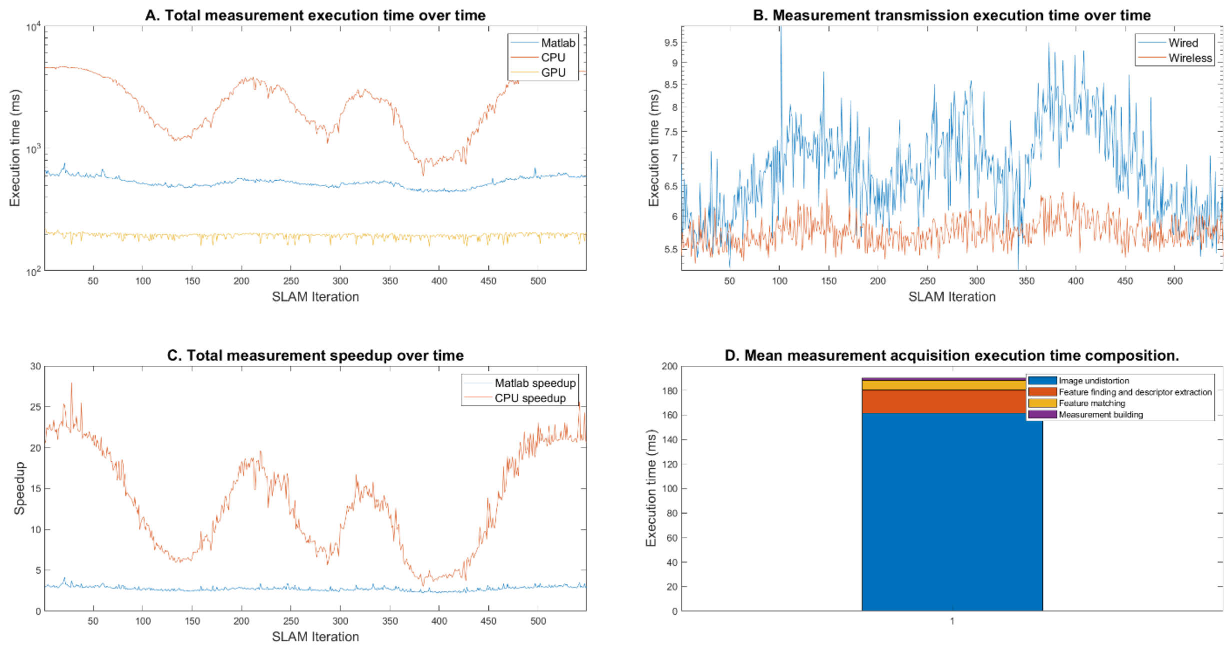


Figure 6.3 A. The execution time of the full landmark measurement acquisition process over time for each implementation. **B.** Execution time of the transmission of the landmark measurements over time using Wi-Fi and Ethernet. **C.** The speedup of the full GPU accelerated measurement acquisition compared to the MATLAB and non-accelerated CPU implementation. **D.** The composition of the mean measurement acquisition execution time.

6.2.2 Landmark matching

The landmark matching algorithm is part of the landmark association of the *bookkeeping* algorithm. Figure 6.4 shows the execution time of the landmark matching algorithm implemented in MATLAB, without GPU acceleration, and with GPU acceleration. Figure 6.4 A shows the execution time of the landmark matching over time. It shows that the execution time of the GPU implementation is at most 10 ms, while the execution time of the CPU implementation is at most 20 ms. The MATLAB implementation has the highest execution time of at most 97 ms. Furthermore, it shows that the execution time increases almost linearly over time, as the amount of landmarks in the pool increases, as can be seen from Figure 6.4 B. This figure also shows that the increase in execution time with the increase in pool size is the highest for the MATLAB implementation, while the GPU implementation has the lowest increase in execution time.

Figure 6.4 C and Figure 6.4 D show the speedup of the accerated matcher. Figure 6.4 C shows the speedup of the accelerated matcher with respect to the CPU matcher and the MATLAB matcher over time. Figure 6.4 D shows speedup of the accelerated matcher over the amount of landmarks in the pool. It can be seen that the speedup increases gradually over time, as the amount of landmarks increases. Furthermore can it be seen that the speedup of the CPU matcher is low if the pool size is low.

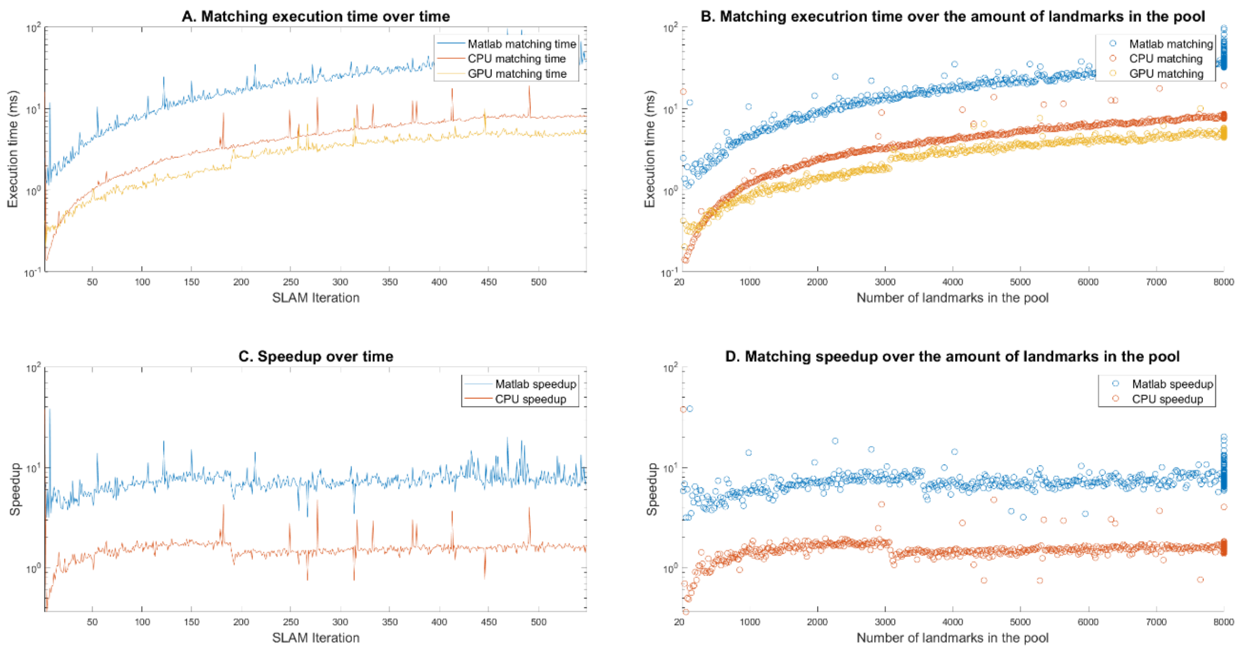


Figure 6.4 A. Comparison of the landmark matching time over time. **B.** Comparison of the landmark matching time over the number of landmarks in the pool. **C.** Speedup of the accelerated matcher over time. **D.** Speedup of the accelerated matcher over the amount of landmarks in the pool.

6.2.3 Bookkeeping

Figure 6.5 shows a comparison of the execution time of the different *bookkeeping* implementations and the speedup of the accelerated implementation together with the execution times of the landmark deletion and addition algorithms over time. It can be seen from Figure 6.5 A, B and D that the total *bookkeeping* time of the CPU implementation is at most 24931 ms with a highest landmark deletion time of 23622 ms and a highest landmark addition time of 1430 ms. The highest total *bookkeeping* time of the MATLAB implementation is 8174 ms with a highest landmark deletion time of 5861 ms and a highest landmark addition time of 2127 ms. The GPU accelerated implementation has a lower total booking time, with its highest value at 4160 ms, with a highest landmark deletion time of 4150 ms and a highest landmark addition time of 1.115 ms.

The speedup shown in Figure 6.5 C shows that the speedup of the CPU implementation is generally lower than the speedup of the MATLAB implementation, until the maximum amount of landmarks in the pool is reached and more landmarks are removed at every iteration, increasing the execution time of the landmark deletion.

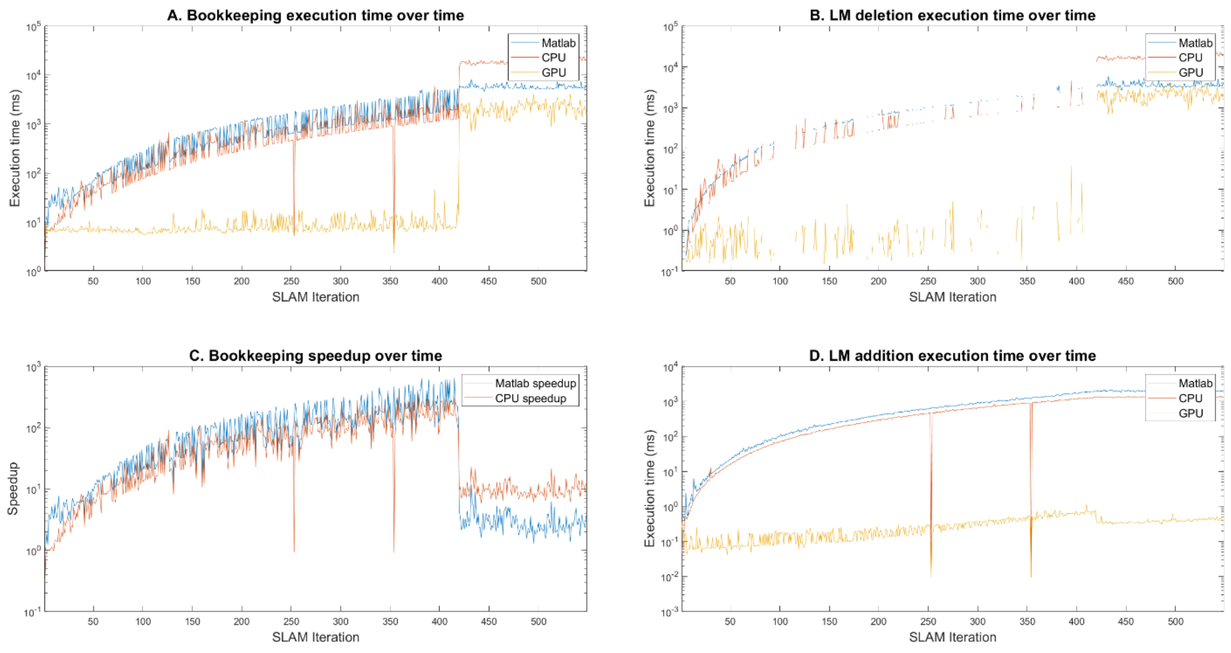


Figure 6.5 A. Comparison of the full *bookkeeping* execution time over time. **B.** Comparison of the landmark deletion execution time over time. **C.** Speedup of the accelerated full *bookkeeping* algorithm over time. **D.** Comparison of the landmark addition execution time over time.

The measured execution times of the landmark deletion and landmark addition are discussed individually.

6.2.3.1 Landmark deletion

Figure 6.5 A shows that the maximum landmark deletion time of the CPU implementation is 23622 ms, while the maximum landmark deletion time of the MATLAB implementation is 5861 ms. The GPU implementation has the lowest maximum landmark deletion time of 4149 ms per landmark.

Figure 6.6 shows the influence of the amount of landmarks in the pool and the amount of deleted landmarks on the execution time of landmark deletion algorithm as well as the speedup of the accelerated landmark deletion algorithm over time and over the amount of landmarks in the pool. Figure 6.6 A shows that the increase in execution time per deleted landmark of the GPU accelerated implementation is small over the increasing number of landmarks in the pool, while the execution time of the MATLAB and CPU implementations of the landmark deletion increases strongly with the pool size. The figure further shows that the execution time of the CPU landmark deletion algorithm is lower than that of the MATLAB counterpart until the maximum number of landmarks in the pool is reached and the number of deleted landmarks increases. Figure 6.6 B shows that the speedup of the CPU and

MATLAB landmark deletion algorithms increase with the number of landmarks in the pool, until the amount of landmarks in the pool reaches its limit and the amount of deleted landmarks increases.

Figure 6.6 C shows the execution time of the landmark deletion algorithms per landmark in the pool over the number of deleted landmarks. It can be seen that the execution time of all implementations increases with the number of deleted landmarks.

Figure 6.6 D shows the speedup of the accelerated landmark deletion algorithm over the number of landmarks in the pool. It can be seen that the speedup is reduced when the maximum pool size is reached.

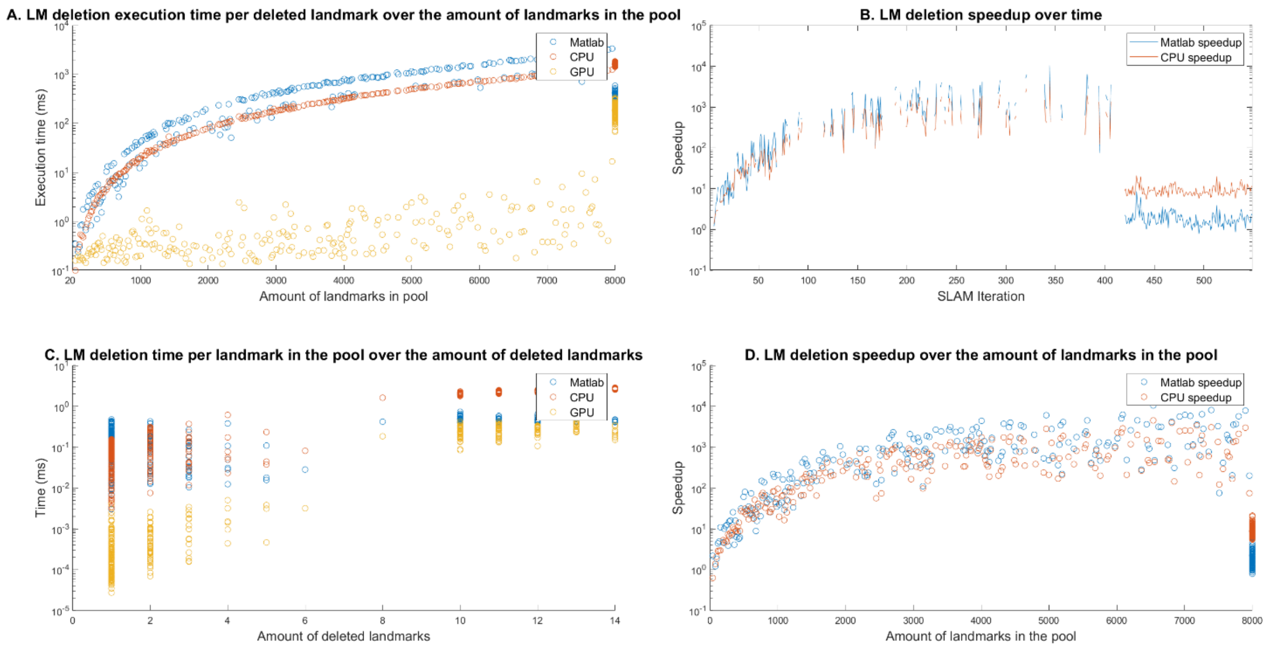


Figure 6.6 A: The landmark deletion execution time per deleted landmark over the number of landmarks in the pool. **B.** Speedup of the accelerated landmark deletion algorithm over time. **C:** The landmark deletion execution time per landmark in the pool over the number of deleted landmarks. **D:** Speedup of the accelerated landmark deletion algorithm over the amount of landmarks in the pool.

6.2.3.2 Landmark addition

Figure 6.7 shows the influence of the amount of landmarks in the pool and the amount of added landmarks on the execution time of the landmark addition algorithm for the various implementations and shows the speedup of the accelerated implementation over time and over the amount of landmarks in the pool. From Figure 6.5 D it can be seen that the maximum execution time of the landmark addition algorithm implemented in MATLAB is 0.2658 ms. The maximum execution time of the landmark addition algorithm executed on the CPU is 0.1788 ms. The execution time of these two implementations increases over time, with the amount of landmarks in the pool as can be seen in Figure 6.7C. The maximum execution time of the GPU accelerated algorithm is 0.0077 ms and increases much slower over time as Figure 6.7 C shows that the landmark addition time also increases slowly over the amount of landmarks in the pool.

Figure 6.7 B shows that the landmark addition speedup increases over time as the amount of landmarks in the pool increases as shown in Figure 6.7 D.

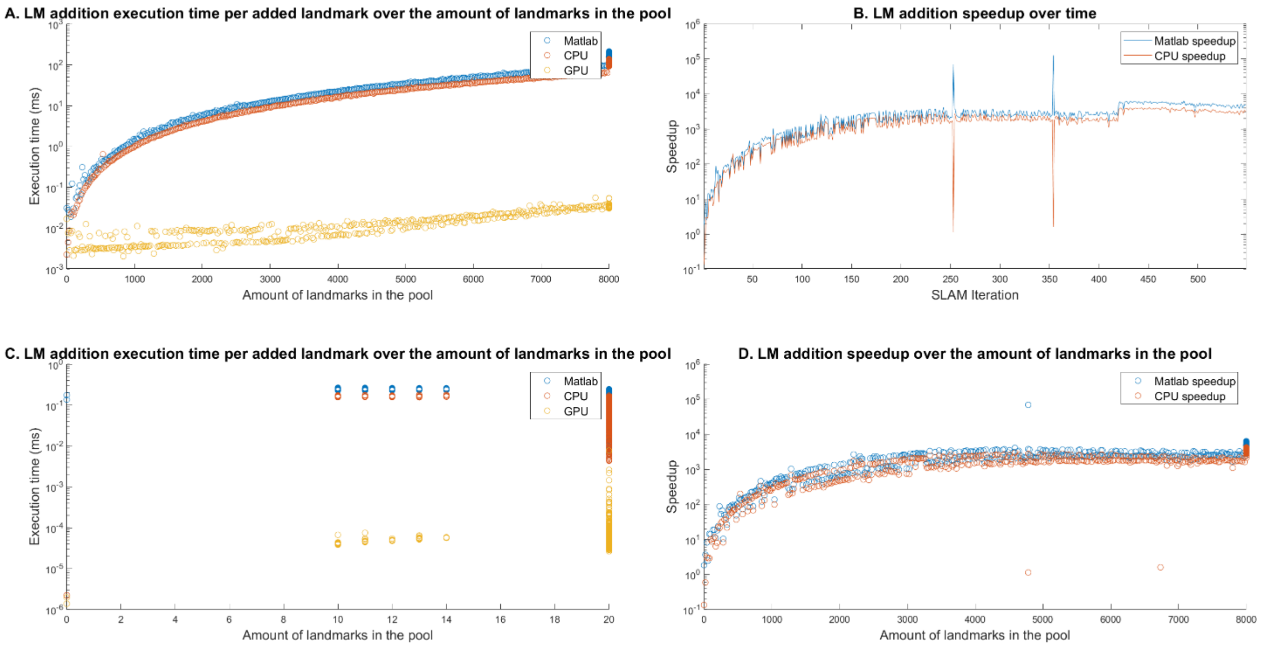


Figure 6.7 A: Comparison of the landmark addition execution time per added landmark over the number of landmarks in the pool. **B.** Speedup of the accelerated landmark addition algorithm. **C:** Comparison of the landmark addition time per landmarks in the pool over the number of deleted landmarks. **D.** Speedup of the accelerated landmark addition algorithm over the number of landmarks in the pool.

6.2.4 ES-EKF computations

Figure 6.8 shows a comparison of the execution time of the Error State EKF computations over time. It shows the time to complete both the *update* and *predict* steps and it shows the execution times of these two steps separately. The figure also shows the total ES-EKF computation time over the number of landmarks in the pool.

It can be seen from Figure 6.8A, C and D that the CPU implementation has a higher computation time than the MATLAB and GPU implementations, of which the GPU implementation has the lowest computation time. The highest computation time of the CPU implementation is 70921 ms, while the highest computation time of the *update* step is 70918 ms and the highest computation time of the *predict* step is 4.556 ms. The highest computation time of the MATLAB implementation is 10992 ms, while the highest computation time of the *update* step is 10899 ms and the highest computation time of the *predict* step is 92.567 ms. The highest computation time of the GPU accelerated implementation is 2287 ms, while the highest computation time of the *update* step is 2287 ms and the highest computation time of the *predict* step is 0.7632 ms. Figure 6.8B shows that the total computation time of the CPU implementation has the highest increase with the number of landmarks in the pool, while the computation times of the MATLAB and GPU implementations only increase slightly with the number of landmarks in the pool.

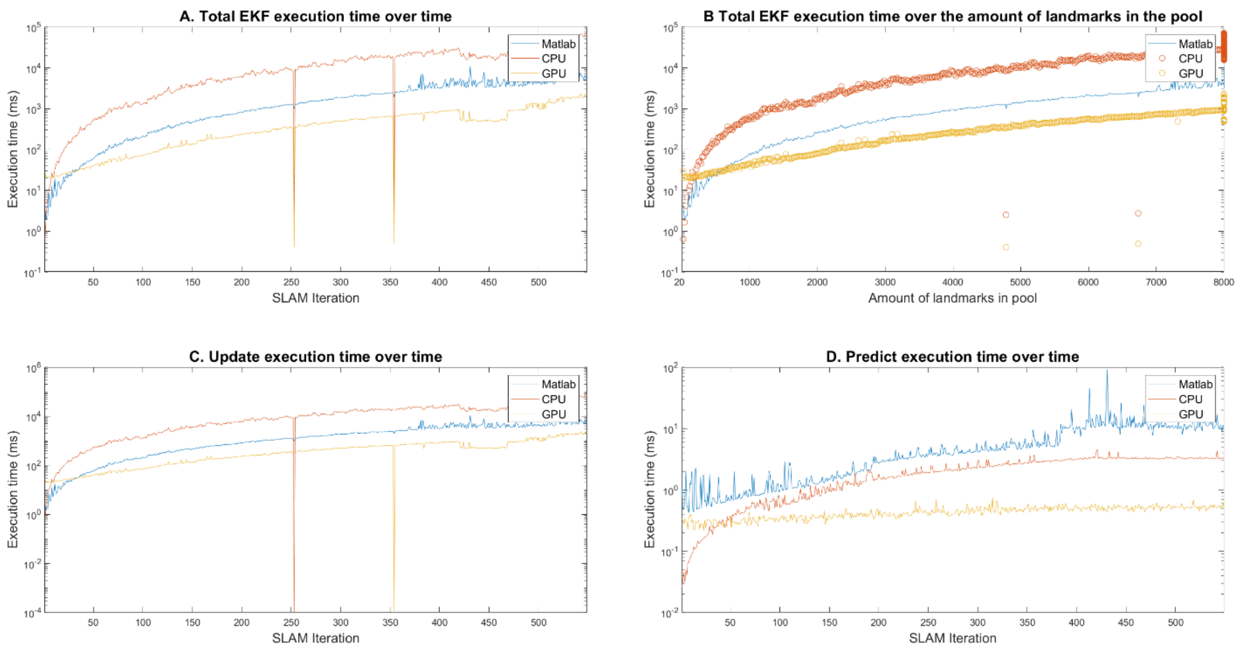


Figure 6.8 A. Comparison of the execution time of the ES-EKF algorithm over time. **B.** Comparison of the execution time of the ES-EKF algorithm over the number of landmarks in the pool **C.** Comparison of the execution time of the ES-EKF Update over time. **D.** Comparison of the execution time of the ES-EKF *predict* over time.

6.2.4.1 Update

Figure 6.9 shows the influence of the number of landmarks in the pool k and the number of visible landmarks l on the computation time of the ES-EKF Update computation time and the speedup of the accelerated algorithm over time and over the amount of landmarks in the pool. Figure 6.9A shows the *update* time per visible landmark over the amount of landmarks in the pool for each implementation. It can be seen that the *update* execution time of the MATLAB and CPU implementations are almost always higher than the *update* execution time of the GPU accelerated implementation, with the CPU implementation almost always having the highest *update* execution time as long as the amount of landmarks in the pool is high enough. From Figure 6.9 D can be seen that as long as there are less than 150 landmarks in the pool the CPU implementation is faster than the GPU accelerated implementation, while the MATLAB implementation is faster as long as the pool size is below 650 landmarks as the speedup is greater than 1.

Figure 6.9C shows the *update* execution time per landmark in the pool over the number of visible landmarks l for each implementation. It can be seen that the execution time per landmark in the pool increases with the amount of visible landmarks.

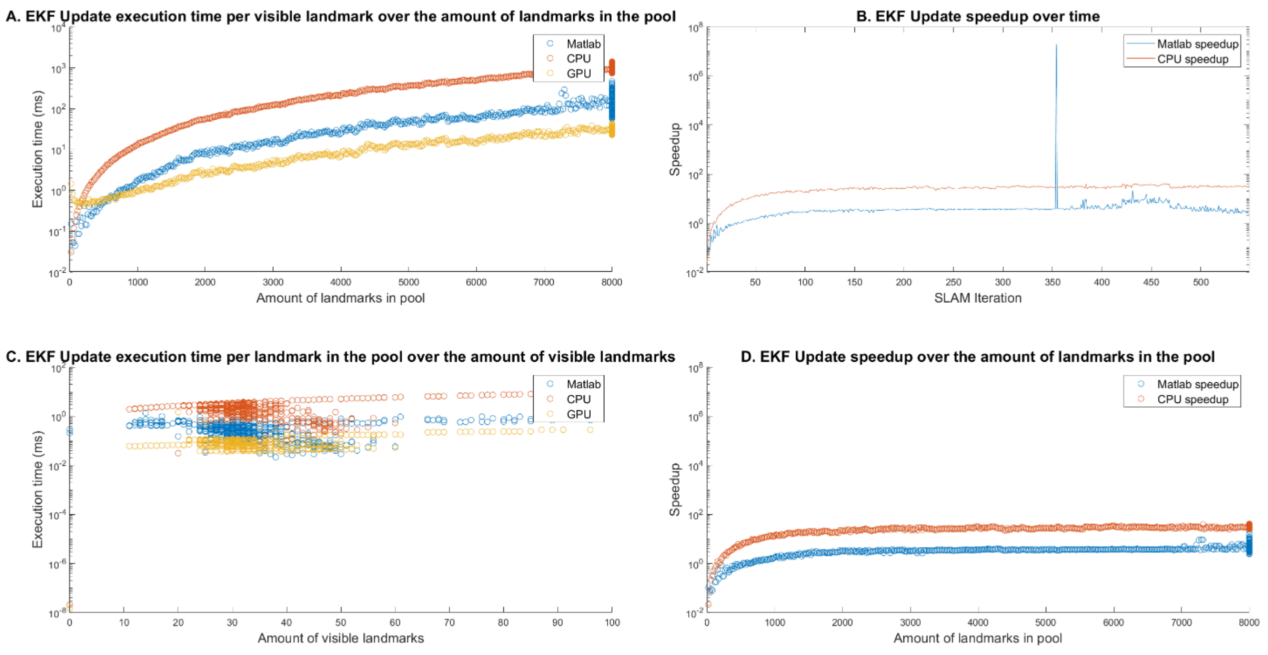


Figure 6.9 A. Comparison of the *update* execution time over the number of landmarks in the pool. **B.** Speedup of the accelerated *update* algorithm over time. **C.** Comparison of the *update* time per landmark in the pool over the amount of visible landmarks. **D.** Speedup of the accelerated *update* algorithm over the number of landmarks in the pool.

6.2.4.2 Predict

Figure 6.10 shows the influence of the amount of landmarks in the pool on the execution time of the *predict* step of the ES-EKF computation and the speedup. Figure 6.10 A shows that the execution time of the *predict* step increases with the amount of landmarks in the pool for all implementations. It can also be seen from Figure 6.10 A and C that the CPU implementation has a lower execution time than the GPU accelerated implementation if there are less than 900 landmarks in the pool. Furthermore does Figure 6.10 B show that the speedup increases over time, as the amount of landmarks in the pool increases as well as shown in Figure 6.10 C.

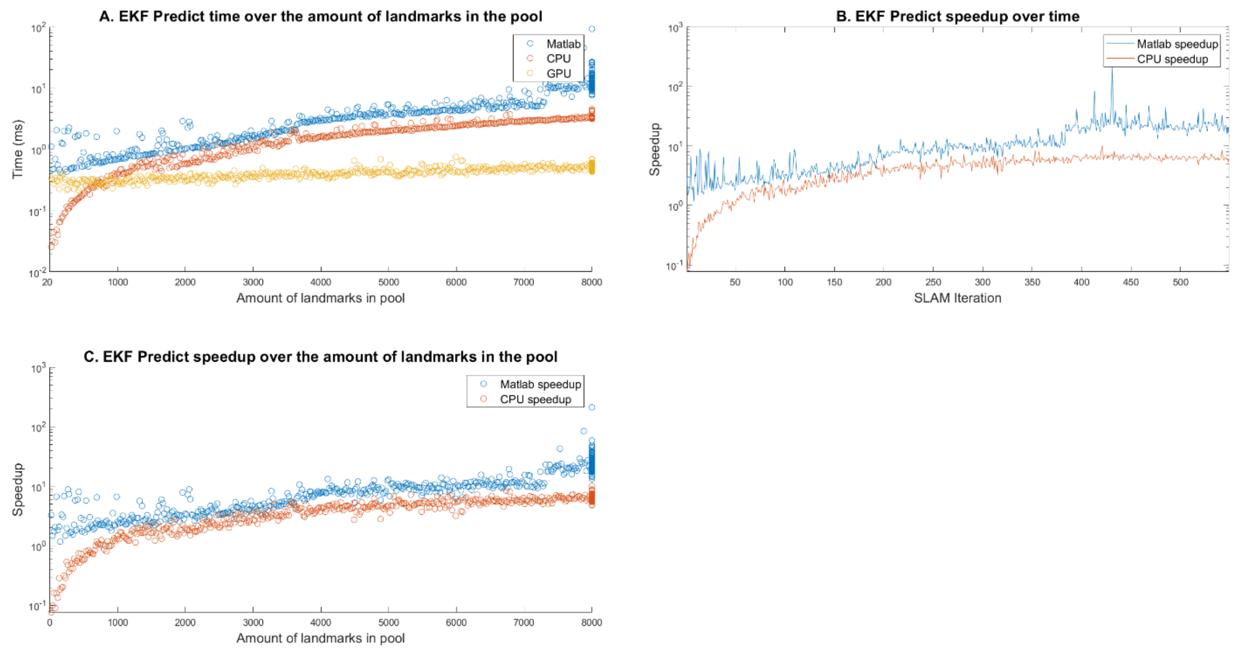


Figure 6.10 A. Comparison of the *predict* time over the number of landmarks in the pool. **B.** Speedup of the accelerated *predict* algorithm over time. **C.** Speedup of the accelerated *predict* algorithm over the amount of landmarks in the pool.

6.2.5 Complete System

Figure 6.11 shows the measured execution times of the complete ES-EKF SLAM implementations: the MATLAB implementation, the CPU implementation without GPU acceleration, and the fully accelerated GPU implementation and the speedup of the GPU accelerated implementation and the effect of the pool size on the execution time and speedup. It can be seen from Figure 6.11 A that the highest measured SLAM execution time of the CPU implementation is 156.8 s, while the highest measured SLAM execution times of the MATLAB and GPU accelerated implementations are 16.44 s and 6.614 s respectively. Figure 6.11 B shows that the execution time of each implementation increases with the pool size.

Figure 6.11 C shows that the speedup for both the CPU implementation and the MATLAB implementation increases, until the maximum pool size is reached, as shown in Figure 6.11 D. Figure 6.11 D also shows that the GPU accelerated implementation is faster than the MATLAB implementation for a pool size larger than 910 landmarks, while the GPU accelerated implementation is faster than the CPU implementation for a pool size larger than 150 landmarks.

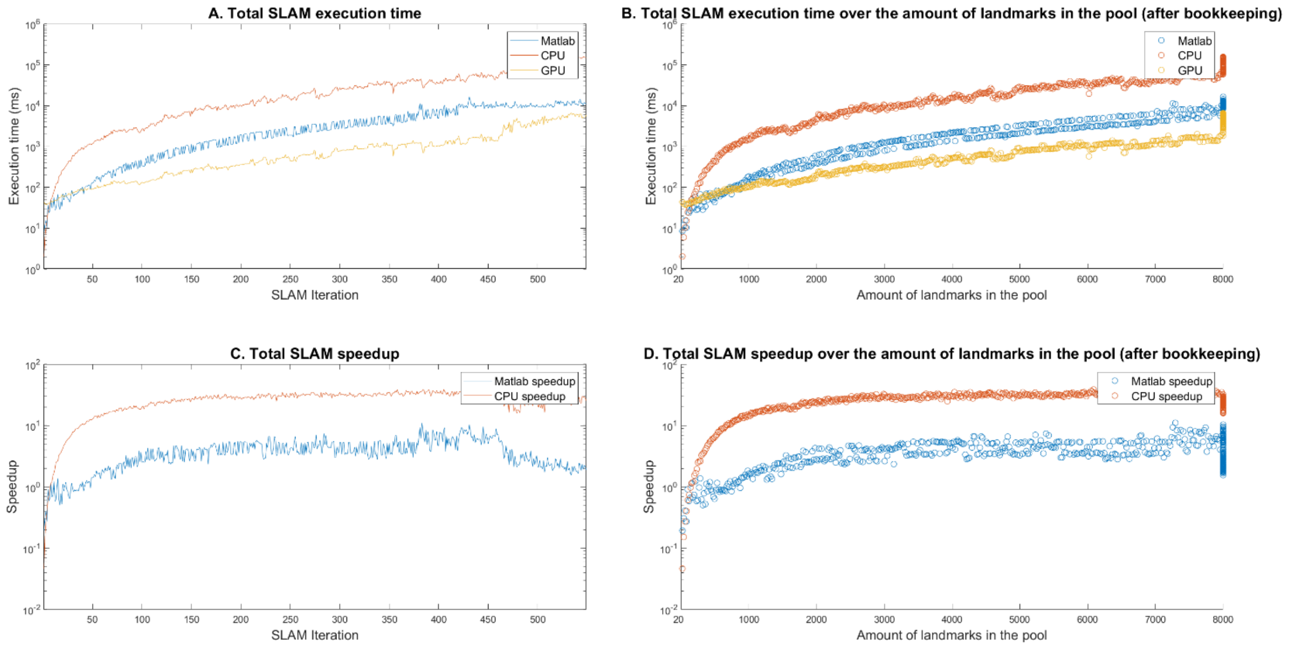


Figure 6.11 A. Comparison of the full ES-EKF SLAM execution time of the different implementations over time. **B.** Complete comparison of the ES-EKF SLAM execution time over the number of landmarks in the pool. **C.** Speedup of the full GPU accelerated ES-EKF SLAM algorithm over time. **D.** Speedup of the full GPU accelerated ES-EKF SLAM algorithm over the amount of landmarks in the pool.

6.3 Worst-case execution time analysis

The real-time analysis looks at the highest execution times for different sizes of the input of the system in order to draw conclusions about the worst-case execution time of the complete system and to choose an input size that ensures real-time performance. As both the landmark measurement acquisition and ES-EKF SLAM parts execute in parallel, the worst-case execution time of a SLAM iteration depends on which part has the highest execution time.

6.3.1 Landmark acquisition

Figure 6.12 shows the influence of the input image size on the execution times of the landmark acquisition algorithm. It can be seen that the image size has an influence on the GPU acceleration algorithm implemented on the NVIDIA Jetson Nano. It can be seen that the execution time increases with the pixel size of the input image. The figure shows that when the size of each image is 384 by 288 pixels, the landmark measurement acquisition takes less than 100 ms and the real-time requirement of performing 10 landmark measurements per second is met as the highest measured execution time of 95.833 ms is less than 100 ms.

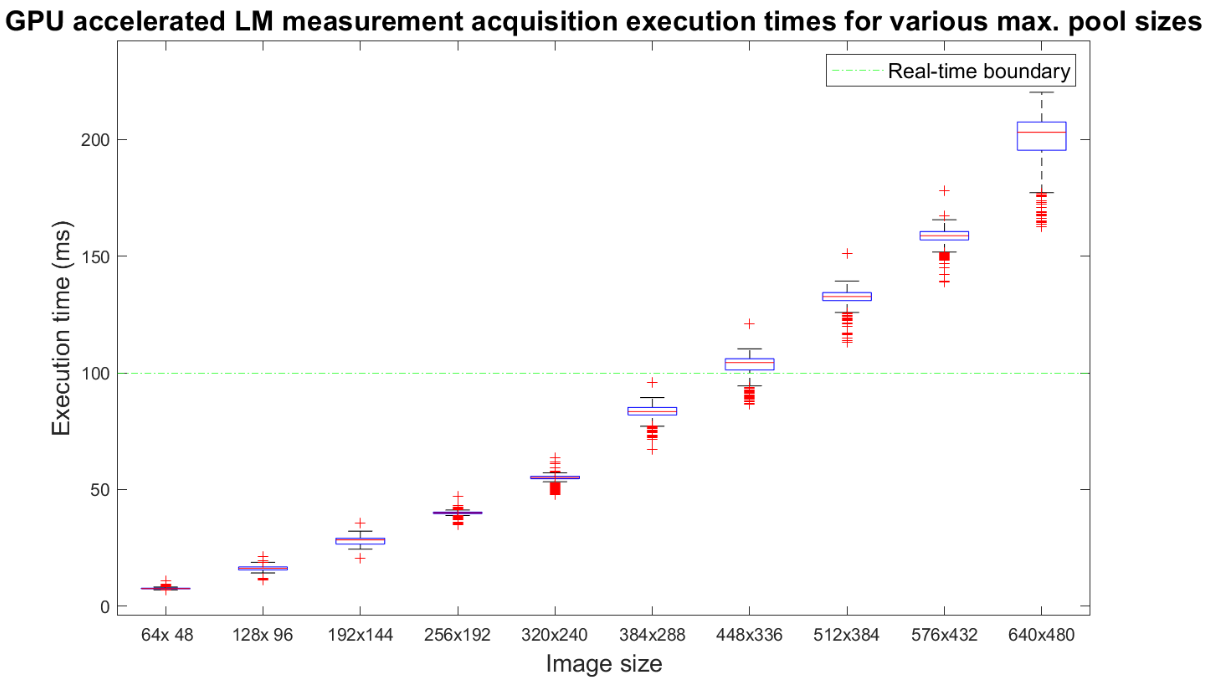


Figure 6.12 Boxplots of the execution times of the landmark measurement acquisition iterations for different input images sizes.

6.3.2 ES-EKF SLAM

As the maximum pool size influences the execution time of the *bookkeeping* algorithm, the ES-EKF *update* and ES-EKF *predict* algorithms, it can be said that the execution time of the EKF SLAM algorithm depends highly on the maximum amount of landmarks in the pool. Therefore, the maximum pool size is the leading parameter to change when optimizing the algorithm by decreasing the problem size.

Figure 6.13 shows the boxplots of execution times of each iteration for different maximum pool sizes. It can be seen that the maximum execution time increases with the maximum amount of landmarks in the pool. The minimum execution time stays constant. The figure also shows that the real-time constraint of at least 10 iterations per second is met as long as the maximum pool size is 1800 landmarks or lower as the maximum measured execution time of 96.8 ms is below the boundary of 100 ms.

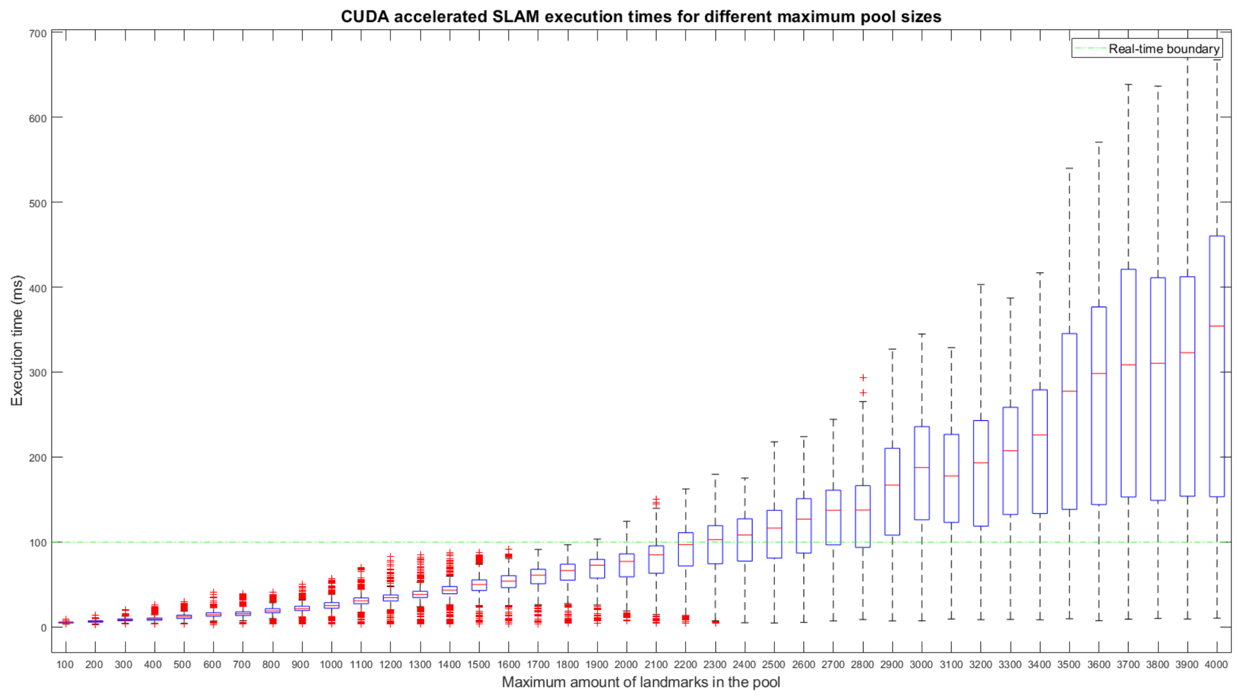


Figure 6.13 Boxplots of the execution times of the ES-EKF SLAM iterations for different maximum state sizes.

7 Discussion

This chapter discusses the results presented in Chapter 6. These results give an insight on the influence of different parameters and input sizes on the execution time of the different landmark measurement acquisition implementations and the ES-EKF SLAM implementations. The results also represent the effectiveness of the GPU acceleration compared to the CPU and MATLAB implementations. The worst-case execution time evaluation also provides an insight on with which input parameters the system fulfills the real-time requirement discussed in Section 3.4.

Before the results can be discussed, it is important to note that the execution times can be influenced by interruptions caused by the operating system. Additionally, thermal throttling can influence the execution times when the CPU or GPU reaches high enough temperatures, causing the clock frequency of the processor and therefore the instruction throughput to decrease.

7.1 Runtime performance

Looking at the speedup of the system, it becomes clear that the GPU acceleration is effective compared to the MATLAB implementation and the implementation without GPU acceleration for both the landmark measurement acquisition and the ES-EKF SLAM computation.

The average speedup of the landmark measurement acquisition on the NVIDIA Jetson nano with respect to the MATLAB implementation is 2.7 meaning that the acceleration on the NVIDIA Jetson nano performs over two times faster than the MATLAB implementation. Unfortunately, the main bottleneck of the landmark acquisition algorithm is the image undistortion part, which is performed on the CPU and takes up the majority of the execution time on the NVIDIA Jetson nano.

Regarding the ES-EKF SLAM implementation, it becomes clear that the GPU acceleration performs faster than the MATLAB and non-accelerated CPU implementations with an average speedup of 3.8 and 26.3 respectively, as long as the number of landmarks is larger than 1000 landmarks in order to fulfill the map density requirement set in Section 3.4. This can be explained by added time needed to copy data between the host processor and the GPU, adding an overhead to the execution time. This overhead becomes smaller as the size of the computations increases with the number of landmarks in the pool.

The execution time of the ES-EKF SLAM algorithm is largely influenced by the number of landmarks in the pool and mainly consists of the landmark deletion execution time and the execution time of the ES-EKF state *update*. This aligns with the findings in Section 3.2, which concludes that these two operations are the most time expensive parts of the ES-EKF SLAM algorithm. As concluded from the worst-case time complexity analysis, the *update* step of the ES-EKF SLAM algorithm had the highest order complexity, which means that the computation time increases with the input size: the number of landmarks in the pool.

The execution time of the landmark deletion algorithm is highly influenced by the number of landmarks in the pool. This is due to the fact that as the size of the state covariance matrix increases, more elements in the matrix need to be manipulated as landmarks that are older are removed due to the construction of the covariance matrix. The execution time is also increased if the number of rows and columns in the state covariance matrix exceeds the number of threads that can be executed concurrently on the GPU, as some computing units need to execute multiple threads sequentially. Also the number of deleted landmarks influences the execution time of the landmark deletion algorithm, as each landmark is removed sequentially, meaning that the algorithm is executed for each deleted landmark.

The landmark addition algorithm is most benefited by the utilized data-level-parallelism on the GPU, as each manipulated element can be processed concurrently. The number of new landmarks therefore has a low influence on the execution time of the landmark addition on the GPU, which is mostly influenced by the number of landmarks in the pool, which again increases execution time if the number of rows and columns in the pool exceeds the number of threads that can be executed in parallel on the GPU, resulting in threads being executed sequentially on some computing units.

The computation of the ES-EKF state *update* and state *prediction* is also significantly accelerated by utilizing the GPU compared to the MATLAB and CPU implementation as long as the pool size is large enough.

The average speedup of the accelerated ES-EKF Update algorithm is 3.7 for the MATLAB implementation and 25.9 for the CPU implementation. The execution time of the ES-EKF *update* is largely influenced by both the number of landmarks in the pool k and the number of visible landmarks l, as is also discussed in Section 3.2, as the worst-case time complexity depends on these values.

Fortunately, the execution time of the ES-EKF *update* algorithm can be further decreased by utilizing the GPU to implement a parallelized algorithm for solving the linear system $GS = CH^T$ for the Kalman Gain matrix G instead of inverting the matrix S and performing the computation $G = C(t|t-1)H^T S^{-1}$. The inversion of S is currently performed on the CPU, requiring the matrix S to be first copied to the host memory, performing the inversion, and then copying the result back to device memory. Implementing the acceleration decreases the data transmission time and utilizes the data-level- and thread-level-parallelism the GPU offers.

Also utilizing the Tensor cores to accelerate the matrix-matrix multiplication can decrease the execution time of the ES-EKF computations. This can be done on the used GPU by using the library developed by Mukunoki, Ozaki, Ogita & Imamura (2020). Their research has confirmed that their implementation results in a major acceleration of FP64 matrix-matrix multiplication using FP16 Tensor cores.

The computation of the ES-EKF *predict* is highly accelerated by utilizing the GPU as well, offering an average speedup of 12.2 for the MATLAB implementation and a speedup of 4.4 for the CPU implementation. The execution time is also impacted by the size of the pool k as indicated by the worst-case time complexity in Section 3.2 as the execution time increases with the number of landmarks in the pool.

As the prediction of the pose part error state vector can be performed concurrently to the *bookkeeping* algorithm and the majority of the ES-EKF *update* algorithm, the low average execution time of 0.7632 ms of the *predict* algorithm should allow for real-time execution using additional sensor measurements. The average execution time means that sensor measurements can be used to perform over 1300 predictions of the camera pose per second.

7.2 Real-Time evaluation

It can be concluded from the real-time evaluation in Section 6.3 that the real-time requirement of executing and processing 10 measurements per second can be met if the input size is limited. As shown in Section 6.3, the worst-case execution time of the ES-EKF SLAM algorithm increases with the maximum amount of landmarks in the pool if the number of visible landmarks is limited. This confirms again the conclusion drawn from the worst-case computational complexity analysis that the execution time is highly dependent on the size of the state vector. Also the execution time of the landmark measurement acquisition increases with the size of the input images.

It is shown that the real-time requirement of acquiring and processing 10 measurements per second is met if the size of each input image is 384 by 288 pixels and the maximum pool size is set to 1800 landmarks. The highest measured execution time of the landmark measurement acquisition is then

95.8 ms and the highest measured execution time of the ES-EKF SLAM algorithm is 96.8 ms. The highest measured measurement transmission time over Wi-Fi is 6.4 ms.

As landmark measurement acquisition, measurement transmission and ES-EKF SLAM computation is performed in parallel, the worst-case execution time is expected to be approximately 96.8 ms, while the time between image acquisition and computation of the result is the combination of the above highest measured execution times: 199 ms, which is the latency of the system.

8 Conclusions

This chapter gives a brief overview of the resulting accelerated ES-EKF SLAM implementation and describes how this implementation fulfills the requirements defined in section 3.4. This chapter is concluded with future recommendations on how more accelerations can be implemented to further increase the execution rate of the ES-EKF SLAM computations and the landmark measurement acquisition.

The resulting system presented in Chapter 5 uses GPUs to utilize data-level parallelism in the ES-EKF SLAM algorithm. A desktop GPU is used to accelerate the computations of the ES-EKF *update*, the matrix-matrix multiplications in the ES-EKF *predict*, and the landmark additions and removal algorithms of the *bookkeeping* step. The landmark measurement acquisition is executed in parallel to the ES-EKF SLAM computations on a single board computer, which can be implemented on the handheld measurement device of the HAPI. The on-board GPU of this single-board computer is used to execute the parallelized computer vision algorithms for landmark finding. Using this model, the system is able to meet the functional requirements regarding execution rate and landmark pool size if the input sizes of the landmark measurement acquisition and the ES-EKF SLAM computations are limited.

The resulting GPU accelerated implementation has an almost quadrupled runtime performance with respect to the MATLAB implementation for a large number of 8000 landmarks in the pool. Unfortunately, a state vector of this size increases the computation time, making the system unable to meet the execution rate requirements discussed in Section 3.4.

As discussed in Section 3.4, the system must perform the landmark measurement acquisition and the ES-EKF SLAM computations with an execution rate of at least 10 Hz, while at least 1000 landmarks are being tracked. The implemented system is able to track 1800 landmarks, while meeting the execution rate requirements of performing 10 ES-EKF SLAM iterations and 10 landmark measurements and per second, as long is the input image size is at most 384 by 288 pixels per stereo image frame. The implemented system therefore meets requirements *FR1*, *FR2*, and *FR8* as defined in Section 3.4.2.2.

Furthermore, the landmark measurement system is required to be handheld and to communicate wirelessly with the SLAM Server. This is achieved by executing the landmark measurement acquisition on a small single-board computer with a relatively low energy consumption, which can be implemented on the handheld device. Wireless communication is performed over Wi-Fi, which adds a low latency to the system for landmark measurement transmission. Finally, as the ES-EKF SLAM implementation is supposed to become a part of the HAPI, the system is required to be expendable for the execution of the handheld perfusion imaging. The implemented system therefore also meets requirements *FR4*, *FR5*, and *FR7* as defined in Section 3.4.2.2.

The resulting system provides a command line interface to start the execution of the landmark measurement acquisition and the ES-EKF SLAM computations with different settings, fulfilling requirements *UR1* and *UR2* as defined in Section 3.4.2.3. As the system is not able to visualize the 3D surface map in real time, requirement *UR3* is not met, but the ES-EKF SLAM results can be stored to be visualized using external software, such as MATLAB.

8.1 Future recommendations

Further accelerations are expected to be possible. As discussed in Chapter 5, multiple accelerations of the ES-EKF computations were not implemented in the resulting system, even though these accelerations were a part of the proposed system design. Additional acceleration possibilities came to light after the runtime performance evaluation of the system.

Accelerations that were not implemented could be used to further accelerate: the matrix-matrix multiplications, the computation of the Kalman gain matrix for the ES-EKF *update*, and to increase the instruction throughput. A parallelized solver of the linear system for the computation of the Kalman Gain matrix can be implemented on the GPU using the CuSolver library. Furthermore, the utilization of Tensor cores to accelerate the matrix-matrix multiplications can be implemented, either by performing the computations using existing libraries for FP64 matrix-matrix multiplications, or using newer hardware that supports these computations out of the box. The instruction throughput can also be increased by using mixed precision for the ES-EKF computations. Finally, further parallelization can be achieved by implementing the parallel execution of EKF state vector prediction for sensor fusion, in order to use the sensor data to increase the accuracy of the ES-EKF SLAM result and to increase the thread-level parallelism in the system.

After the runtime performance analysis, an acceleration opportunity came to light which could further accelerate the execution of the landmark measurement acquisition. This can be done by parallelly compensating the 2D pixel locations of the found landmarks for the lens deformation, instead of undistorting the full images beforehand. This decreases the number of operations that are executed for the compensation of lens distortion, as the number of landmarks is much lower than the number of pixels in a stereo image pair.

Even though the implementation of additional sensors was a functional requirement (*FR3* and *FR6)*, they were not implemented. Such sensors can be implemented in the future to increase the measurement rate of the translational and angular velocity. As discussed by Shah (2020), implementing these sensors is also expected to result in a more accurate result of the ES EKF-SLAM algorithm.

The results of the accelerated ES-EKF SLAM can be further improved to increase accuracy. A feature descriptor and matcher can be used that results in more accurate feature matches and landmark associations. This will decrease mismatches, resulting in more accurate 3D landmark estimations and therefore a more accurate ES-EKF SLAM result.

Finally, UI requirements that are not met by this implementation of the system could be met during the implementation of the ES-EKF SLAM algorithm and landmark measurement acquisition on the HAPI. On-device controls will allow the operator to control the HAPI handheld. Cameras could be implemented to capture the images for landmark measurement. And a viewer can be implemented to visualize the ES EKF-SLAM results in real-time if the SLAM Server transmits the results back to the handheld device.

Acknowledgements

After finishing this thesis, I will look back fondly at my time working on it. This is mainly possible thanks to the people that have provided their guidance, support, advice, or a coffee break, helping me through a masters' thesis during an already difficult time due to the COVID-19 pandemic.

First, I want to thank dr.ir. Ferdi van der Heijden for providing his supervision. You have provided me with the advice, knowledge, and structure throughout the project. I appreciate the guidance to not let me diverge, keeping the workload reasonable and by not letting me deviate from the goal too much. This has helped me develop further, even at this late stage of my student career.

I would also like to thank dr.ir. Jan Broenink for providing me with clear feedback and justified criticism. You have helped me take a better look at my work and looking for ways on how to improve it. You had my best interest in mind, and I appreciate that.

Next, I want to thank the people from BMPI; prof.dr.ir. Wiendelt Steenbergen, Tom Knop, Ata Chizari, Tommie Verouden and Wilson Tsong. I highly appreciated the bi-weekly meetings through which I had a chance to see and talk with people while working from home. I appreciate everyone listening to my input and I want to thank you for letting me be a part of a project with such a direct aim at helping people.

My thanks and appreciation also go to the others who have worked on this project. Mainly Nimish Shah, who helped my understand the ES EKF-SLAM algorithm. Also special thanks to Mike Evers and Gideon Kock, who offered me a chance of occasional coffee breaks while working from home to talk about life and 3D printers.

I would also like to thank my family for supporting me throughout my education and supporting my choices in my career path. You probably all saw it coming. I also want to thank my friends for providing a distraction and for rubber ducking by listening to me ramble for the past year. Even though many do not know how to code, you have helped solving a surprising amount of bugs!

Finally, I want to thank my girlfriend, Benedetta. I do not know how I could have done this without you. You have managed to keep me sane throughout the painful bug fixing processes, late night coding and writing. Of course, your report writing knowledge, listening ear and delicious cooking is always of great help!

A Appendix 1: Optical Flow for Handheld Perfusion Imaging

A.1 Introduction

To perform robust motion measurement for compensation of motion artifacts for handheld perfusion measurement, a system has been proposed using a gyroscope for the measurement of rotational motion, and optical flow measurement for the estimation of translational motion. Since many different algorithms exist for the computation of optical flow which have a different performance, it is of importance that an algorithm is selected which performs best when applied to the HAPI system.

Quantitative analysis of various optical flow algorithms has been performed in the past by Barron, Fleet and Beauchemin (Barron, Fleet, & Beauchemin, 1994) with a focus on the accuracy and the density of the velocity estimates produced by the optical flow algorithms. They have tested the optical flow algorithms on real and synthetic image sequences for which the 2D motion fields were known and concluded that the differential-based algorithm from Lucas and Kanade and the phase-based algorithm by Fleet and Jephson were the most reliable.

However, since the analysis of Barron, Fleet and Beauchemin, new optical flow algorithms have been proposed such as the algorithm proposed by Farnebäck (Farnebäck, 2003). The analysis of Barron, Fleet and Beauchemin also focused on the resulting motion field, instead of the conversion of the motion field to the motion of a single measured surface, which is relevant in the context of the HAPI device. This calls for a new analysis of the performance of various optical flow algorithms with application to handheld perfusion imaging.

This research provides an analysis of the performance of available optical flow algorithms, with the goal of selecting the most reliable algorithm for translational motion measurement for the HAPI device. As the main software development tool for the HAPI device is MATLAB, the optical flow computation methods which are provided by MATLAB are focused on during this research. As the implementation of the algorithms requires knowledge of their parameter configuration to ensure optimal performance for each algorithm, a theoretical analysis of the optical flow algorithms is provided, followed by a description of how they are implemented by MATLAB. Then, the algorithms are experimentally evaluated in two experiments where real image sequences of moving objects with a known velocity are used and compared against an implementation of an optical flow algorithm which is previously proposed for the HAPI project. This algorithm is the affine flow algorithm, which is based on the Lukas-Kanade algorithm and automatically converts the optical flow into a displacement in pixels/frame.

A.2 Optical flow algorithms

As explained, the optical flow algorithms that are analyzed are the algorithms which are provided by MATLAB. This section provides a theoretical analysis of the algorithms provided by MATLAB 2020 to calculate optical flow. The goal of this analysis is to have a better understanding of the different algorithms and their implementation in software. The discussed algorithms are listed below:

- Farnebäck
- Lukas-Kanade
- Horn-Schunck
- Lukas-Kanade Derivative of Gaussian

A.2.1 Optical Flow Equation

The estimation of optical flow is based on two assumptions:

- The constant brightness assumption: the color values of a surface patch do not change when the surface patch moves (Barron, Fleet, & Beauchemin, 1994):
$$f(x(t), y(t), t) = constant \tag{1}$$
- The continuity assumption: the motions of two neighboring surface patches are almost the same.

In a grey-scale color domain, a sequence of frames can be represented by $f(x, y, t)$ where (x, y) are the pixel coordinates and t is time. Suppose a surface patch is projected on the image plane at position $[x(t), y(t)]$. The velocity of this surface patch projected on the image plane then becomes:

$$\mathrm{v} = \left[\frac{dx}{dt}\ \frac{dy}{dt}\right]^T \tag{2}$$

This definition can be split up in a horizontal and a vertical velocity:

$$\mathrm{v} = \ [u(t)\ v(t)]^T \text{ with } u(t) = \frac{dx}{dt} \text{ and } v(t) = \frac{dy}{dt}.$$

Now if we assume that $f(x, y, t)$ is a differentiable function and we combine this assumption with the constant brightness assumption, we can derive that the derivative of $f(x, y, t)$ over time is zero:

$$\frac{d}{dt} f(x(t), y(t), t) = 0 \tag{3}$$

The chain rule of derivation can be applied to this Equation to split the Equation up in multiple partial derivatives:

$$\frac{df(x, y, t)}{dx}\frac{dx}{dt} + \frac{df(x, y, t)}{dy}\frac{dy}{dt} + \frac{df(x, y, t)}{dt} = 0 \tag{4}$$

This Equation can be simplified:

$$f_x u + f_y v + f_t = 0 \tag{5}$$

$$\text{with } f_x = \frac{df(x,y,t)}{dx},\ f_y = \frac{df(x,y,t)}{dy},\ f_t = \frac{df(x,y,t)}{dt}$$

This Equation is known as the optical flow Equation (OFE). Where f_x is the derivative of the pixel brightness over the x-direction of the frame, f_y is the derivative of the pixel brightness over the y-direction of the frame, and f_t is the derivative of the pixel intensity of the frame over time.

Solving this Equation results in the pixel location differences u and v for each pixel in the frame, and thus provides the optical flow between two frames.

A.2.2 Horn-Schunck

The Horn-Schunck method (Barron, Fleet, & Beauchemin, 1994) assumes that the movement over the whole image is smooth. This method tries to estimate a velocity field $[u\ v]^T$ that minimizes the following Equation:

$$E = \iint \left(f_x u + f_y v + f_t\right)^2 dxdy + \alpha \iint \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 dxdy \tag{6}$$

This Equation minimizes two separate integrals which define the brightness and smoothness constraints:

$$\iint \left(f_x u + f_y v + f_t\right)^2 dxdy \tag{7}$$

The constant brightness constraint

$$\iint \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 dxdy \tag{8}$$

The smoothness constraint

$\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}$, and $\frac{\partial v}{\partial y}$ are the derivatives of the horizontal and vertical velocity respectively over the horizontal and vertical directions of the image.

In this Equation, α scales the smoothness constraint. This means that the bigger one chooses for α, the smoother the motion is assumed to be over the image, while a smaller value for α assumes a less smooth motion (e.g., when the frame consists of elements that move different from each other). This value is set to 1 by default, however, in another implementation by Barron et. Al. (Barron, Fleet, & Beauchemin, 1994) this value was set to 0.5, while Horn and Schunck initially proposed a value of 100 (Horn & Schunck, 1981).

The Equation above is solved for u and v using the following iterative Equations:

$$u_{x,y}^{k+1} = \bar{u}_{x,y}^{k} - \frac{f_x\left[f_x \bar{u}_{x,y}^{k} + f_y \bar{v}_{x,y}^{k} + f_t\right]}{\alpha^2 + {f_x}^2 + {f_y}^2} \tag{9}$$

$$v_{x,y}^{k+1} = \bar{v}_{x,y}^{k} - \frac{f_y\left[f_x \bar{u}_{x,y}^{k} + f_y \bar{v}_{x,y}^{k} + f_t\right]}{\alpha^2 + {f_x}^2 + {f_y}^2} \tag{10}$$

Here $u_{x,y}^{k}$ and $v_{x,y}^{k}$ are the velocities for the pixel at location (x, y). $\bar{u}_{x,y}^{k}$ and $\bar{v}_{x,y}^{k}$ are the average velocities of the four neighboring pixels (as shown in the figure below) resulting from the previous iteration. These average velocities are 0 if the current iteration is the first iteration ($k = 0$).

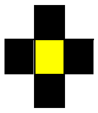


Figure A.1 Visualization of neighboring pixels.

k denotes the iteration. The maximum number of iterations is set to 10 by default. A higher value for k is used to estimate the optical flow for low velocities. When the maximum for k is reached, the algorithm stops.

Another stopping condition in this algorithm is met when the minimum absolute velocity difference is reached. This is the difference between the velocity in either the horizontal or vertical direction at a location (x, y) in the frame between two consecutive iterations. This value is set to 0 by default and if it is desired to use the maximum number of iterations it needs to be set to 0. The lower this value is set, the lower the velocity of the bodies in the frame is assumed to be.

A.2.2.1 MATLAB implementation

The MATLAB implementation for the Horn-Schunck algorithm by solving for u and v using the Equations above works as follows (Object for estimating optical flow using Horn-Schunck method, n.d.):

1. Compute f_x and f_y using the Sobel convolution kernel for each pixel in the first image. This kernel is defined as $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ to get the spatial derivative of the image in the y-direction and is transposed to get the derivative in the x-direction.
2. Compute f_t by subtracting the first image from the second image.
3. Iteratively solve for u and v following the Equations above, where the average velocity is set to 0 for $k = 0$ and compute the average velocity for the following iterations using the kernel $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

A.2.3 Lucas-Kanade

The Lucas-Kanade method solves the optical flow Equation by first dividing the frame into smaller sections and assuming a constant velocity in each section (Barron, Fleet, & Beauchemin, 1994). Then a weighted, least-square fit of the optical flow Equation to a constant model for u and v in each section **Ω**. This fit is achieved by minimizing

$$\sum_{x,y \in \Omega} W^2 [f_x u + f_y v + f_t]^2 \tag{11}$$

Where each pixel at location (x, y) is within section **Ω**. W denotes a window function which emphasizes the constraints at the center of each section.

The Equation above does not contain the smoothness constraint like the Horn-Schunck method does. The Lucas-Kanade method includes the smoothness constraint by assuming that the motion is smooth within each section of the frame, but not over the whole frame.

The solution to this Equation is as follows:

$$\begin{bmatrix} \sum W^2 f_x^2 & \sum W^2 f_x f_y \\ \sum W^2 f_y f_x & \sum W^2 f_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum W^2 f_x f_t \\ \sum W^2 f_y f_t \end{bmatrix} \tag{12}$$

Before solving this Equation for u and v, the window function is applied to the images. This window function smooths the derivatives of the input image using a gaussian filter to decrease the effects of aliasing and quantization in the images. This is done by using the kernel $[1 \quad 4 \quad 6 \quad 4 \quad 1]/16$ for the x-direction, and its transposed form for the y-direction. This also means that the size of each section **Ω** is 5 by 5 pixels.

Finally, solving the Equation above depends on the eigenvalues of:

$$A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sum W^2 f_x^2 & \sum W^2 f_x f_y \\ \sum W^2 f_y f_x & \sum W^2 f_y^2 \end{bmatrix} \tag{13}$$

These eigenvalues λ_1 and λ_2 are compared to a set threshold to reduce noise:

- Are both eigenvalues below the threshold, the values for u and v are calculated using Cramer's rule.
- Is λ_1 smaller than the threshold, a normal velocity estimate is computed.
- Are both eigenvalues below the threshold, the optical flow u and v is 0.

This threshold is set to 0.0039 by default. Increasing the value will result in a resulting flow which has less motion elements, as smaller motions are ignored.

A.2.3.1 MATLAB implementation

The MATLAB implementation of the algorithm works as follows (Object for estimating optical flow using Lucas-Kanade method, n.d.):

1. Compute f_x using the kernel $[-1 \quad 8 \quad 0 \quad -8 \quad 1]/12$ and f_y using the transposed form of the same kernel.
2. Compute f_t by subtracting the first image from the second image.
3. Apply the window function by smoothing the derivatives f_x, f_y and f_t using the kernel $[1 \quad 4 \quad 6 \quad 4 \quad 1]/16$ for the x-direction, and its transposed form for the y-direction.
4. Compute the eigenvalues of A using the following Equation:
$$\lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1,2$$
5. Compare the resulting eigenvalues to a threshold τ
    a. If $\lambda_1 \geq \tau$ and $\lambda_2 \geq \tau$: Solve Equation 12 using Cramer's rule.
    b. If $\lambda_1 \geq \tau$ and $\lambda_2 < \tau$: normalize the gradient flow to compute u and v.
    c. If $\lambda_1 < \tau$ and $\lambda_2 < \tau$: $u = 0$ and $v = 0$.

A.2.4 Lucas-Kanade with Derivative of Gaussian Filter

The Lucas-Kanade method has been expanded to compute f_t using a derivative of Gaussian filter (Object for estimating optical flow using Lucas-Kanade derivative of Gaussian method, n.d.). This algorithm still solves Equation 11 using Equation 12 in a way similar to the vanilla Lucas-Kanade method, but differs in the way it computes f_x, f_y and f_t.

The derivative of Gaussian method does not just perform spatial filtering on the image to compute derivatives f_x and f_y, but first performs temporal filtering on the image using a gaussian filter. This means that pixel values are smoothened over time. This depends on a set number of frames, which also defines the standard deviation and size of the filter. Increasing the number of frames, results in an optical flow which is less sensitive to high frequency movements. The default setting for this option is set to 3 frames.

Then a gaussian filter and derivative of a gaussian filter are used to filter the previously filtered image in the spatial domain, resulting in f_x and f_y. This spatial filter has a length dependent on the standard deviation which can be specified as well, which influences the smoothness of the spatial gradient. This setting is defined as the *imageFilterSigma* property, which is set to 1.5 by default. Increasing this value results in a smoother f_x and f_y, with smoother edges.

Then f_t is computed between the current image and the previous images. This is done by first performing temporal filtering of the current image and previous images based on the number of images that are kept in the memory based on the defined number of frames. This filter uses the derivative of a gaussian. The result is then smoothed using the same spatial filter used to smooth f_x and f_y.

Then f_x, f_y and f_t are smoothed using a spatial filter. This gaussian filter is defined by the setting `GradientFilterSigma`, which defines the standard deviation and size of the kernel. The default value for this setting is 1. Increasing this value means that the result is less sensitive to rapid change of motion.

A.2.4.1 MATLAB implementation

The MATLAB implementation of the algorithm works as follows:

1. Compute f_x and f_y by first performing temporal filtering based on previous images using a gaussian filter and then use a gaussian filter and a derivative of gaussian filter to perform special filtering and smoothing the image.
2. Compute f_t by temporal filtering the current image and previous images using a derivative of gaussian filter and then performing spatial filtering using a gaussian filter.
3. Smooth f_x, f_y and f_t using a gaussian filter.
4. Compute the eigenvalues of A using the following Equation:
$$\lambda_i = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1{,}2$$
5. Compare the resulting eigenvalues to a threshold τ
   a. If $\lambda_1 \geq \tau$ and $\lambda_2 \geq \tau$: Solve Equation 12 using Cramer's rule.
   b. If $\lambda_1 \geq \tau$ and $\lambda_2 < \tau$: normalize the gradient flow to compute u and v.
   c. If $\lambda_1 < \tau$ and $\lambda_2 < \tau$: $u = 0$ and $v = 0$.

A.2.5 Farnebäck

The Farnebäck method for calculating optical flow assumes that an image can be represented by a second order polynomial Equation with matrix *A*, vector *b*, and scalar *c* (Farnebäck, 2003):

$$f(x) = x^T Ax + b^T x + c \tag{14}$$

Using this assumption, a second image with a displacement *d* can be defined as:

$$f(x-d) = (x-d)^T A(x-d) + b^T(x-d) + c \tag{15}$$

This Equation can be rewritten to define an Equation for image 2 of similar form as Equation 14:

$$\begin{aligned} f_2(x) = f(x-d) &= (x-d)^T A_1(x-d) + b_1^T(x-d) + c_1 \\ &= x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - b_1^T d + c_1 \\ &= x^T A_2 x + b_2^T d + c_2 \end{aligned} \tag{16}$$

With

$$A_2 = A_1 \tag{17}$$
$$b_2 = b_1 - 2A_1 d \tag{18}$$
$$c_2 = d^T A_1 d - b_1^T d + c_1 \tag{19}$$

From Equation 18 we can solve for the displacement *d* as follows:

$$2A_1 d = -(b_2 - b_1) \tag{20}$$

$$d = -\frac{1}{2}A_1^{-1}(b_2 - b_1) \tag{21}$$

The global polynomial assumption is then replaced by a local assumption based on second order polynomials to form a displacement for each pixel of the image. This results in the coefficients $A_1(x)$, $b_1(x)$, and $c_1(x)$ for the first image and $A_2(x)$, $b_2(x)$, and $c_2(x)$ for the second image.

Because the problem is now local, *A* and *b* are dependent on the location *x*:

$$A(x) = \frac{A_1(x) + A_2(x)}{2} \tag{22}$$

$$\Delta b(x) = -\frac{1}{2}\big(b_2(x) - b_1(x)\big) \tag{23}$$

Substituting Equation 22 and Equation 23 into Equation 20 then gives the following:

$$A(x)d(x) = \Delta b(x) \tag{24}$$

Here, *d(x)* indicates the local displacement. So far, the brightness constraint of the optical flow problem is being taken in account but solving Equation 24 for each pixel can result in noise. So, the smoothness constraint is now introduced, using the assumption that the motion field contains relatively smooth motion, meaning that the motion varies slowly over the frame. This makes it possible to solve for $d(x)$ by minimizing the following function for a neighborhood I defined by a window function $w(\Delta x)$:

$$\sum_{\Delta x \in I} w(\Delta x)||A(x + \Delta x)d(x) - \Delta b(x + \Delta x)||^2 \tag{25}$$

This minimum is obtained for $d(x)$ by:

$$d(x) = \left(\sum w\Delta A^T A\right)^{-1} \sum wA^T \Delta b \tag{26}$$

The minimum value of Equation 25 is given by:

$$e(x) = \left(\sum w\Delta b^T \Delta b\right) - d(x)^T \sum wA^T \Delta b \tag{27}$$

This means that $A^T A$, $A^T \Delta b$, and $b^T \Delta b$ can be computed for each pixel and can then be averaged to solve for the displacement $d(x)$.

The assumption that the local polynomials are the same for both images except for a displacement holds for a smooth and small motion but might not hold for motions that vary highly. A priori knowledge can therefore be used to increase the accuracy of local displacement computations. If this a priori knowledge about the displacement field, it can be used to compare the polynomial at location x in the first image to the polynomial $x + \tilde{d}(x)$ in the second image, where $\tilde{d}(x)$ is the a priori displacement field. It is then necessary to estimate the displacement between the a priori estimate and the real displacement.

The a priori knowledge is implemented in Equation 22 and Equation 23 as follows:

$$A(x) = \frac{A_1(x) + A_2\big(x + \tilde{d}(x)\big)}{2} \tag{28}$$

$$\Delta b(x) = -\frac{1}{2}\left(b_2\left(x + \tilde{d}(x)\right) - b_1(x)\right) + A(x)\tilde{d}(x) \tag{29}$$

Now $d(x)$ can be calculated as follows:

$$d(x) = A^{-1}\Delta b(x) \tag{30}$$

This way, an iterative algorithm can be implemented to compute a precise displacement by minimizing the difference between the displacement and the a priori displacement for each iteration. Unfortunately, if the first iteration results in a displacement which is too big and the displacement difference with the second iteration cannot be minimized, the resulting displacement map is less accurate. This issue is solved by implementing iterations with multiple scales as shown in Figure A.2: the first iteration uses a low-resolution version of the image and passes the displacement knowledge down to the next iteration which uses images of higher resolution to refine the displacement.

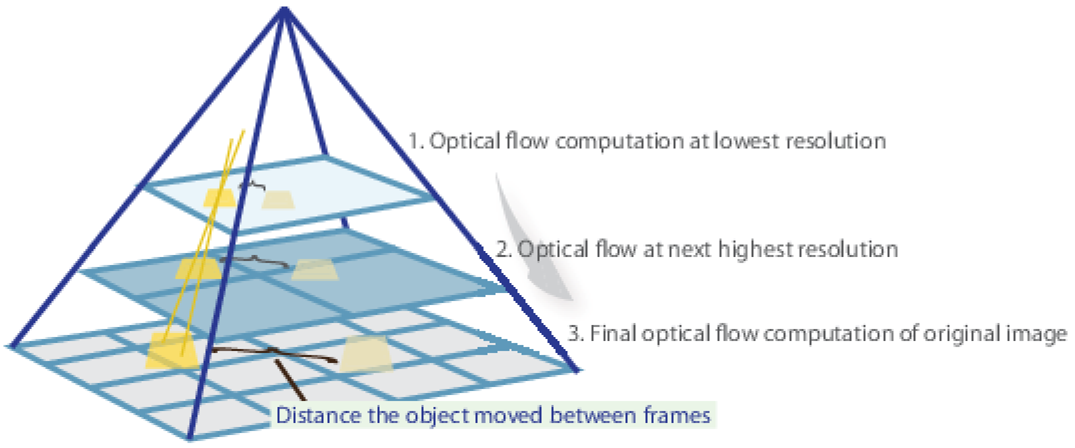

1. Optical flow computation at lowest resolution
2. Optical flow at next highest resolution
3. Final optical flow computation of original image
Distance the object moved between frames

Figure A.2 diagram showing the iterative manner at which the displacement map is refined (Object for estimating optical flow using Farneback method, n.d.).

A.2.5.1 MATLAB implementation

MATLAB uses the OpenCV implementation of the Farnebäck algorithm. This algorithm implements the iterative pyramid as described above, iteratively minimizing Equation 25.

MATLAB provides several settings to customize the behavior of the algorithm, including the size and shape of the pyramid, window size, and other settings (Object for estimating optical flow using Farneback method, n.d.).

Starting with the settings related to the pyramid, MATLAB allows for configuration of the number of layers in the pyramid and the scale of each layer with respect to the lower level. The number of levels is set to 3 by default. Increasing this value increases the accuracy if the refinement of the resulting optical flow, but also increases computation time. Each level in the pyramid has a certain scale with respect to the level below. This setting is set to 0.5 by default, resulting in the next level to be half the size of the image at the level below.

The implementation also performs multiple iterations of displacement computations at each level, using the estimated displacement as a priori knowledge for the second iteration. The number of iterations can be changed and is set to 3 by default. Increasing this number results in more accurate displacement values resulting from each pyramid level, but also increases computation time.

The size of neighborhood I can also be set. The default value is 5 and increasing the value results in a smoother motion field, as motion is assumed to be smooth in a larger area.

Finally, after each iteration when a flow is computed within a pyramid level, the resulting flow is filtered using a gaussian filter, resulting in a smoother motion field. The size of this filter can be set, and larger values increase the robustness to image noise and increases detection of faster motion. The default value for this setting is 15.

A.3 Method

The experimental analysis of the performance of the discussed optical flow techniques in the context of the HAPI system is separated in two experiments. During the first experiment, all optical flow algorithms are configured for optimal performance on an image sequence of a surface with well-defined features which moves with various known static velocities in the horizontal direction. The measured surface is a printed image of a QR code. Then the performance of the algorithms is analyzed and compared in order to select the two best performing algorithms.

The performance of the best two optical flow algorithms is then analyzed and compared to a method for optical flow computation which was previously implemented on the HAPI system. Two image sequences are used for this analysis; one image sequence containing images of the same surface which was used during the first experiment and an image sequence containing the surface of a human hand. The image sequence has an equal, known, realistic motion.

A.3.1 Used Materials

The materials used during both experiments are the following.

- Computer with the following software:
    - MATLAB 2019
    - Zaber Motion Library for MATLAB
    - Basler Camera Driver for MATLAB
    - Image acquisition Toolbox
    - Experiment MATLAB script
- Zaber X-LHM200A-E03 Motorized Linear Stage
- Basler aca1920-40 uc camera with 12 mm lens with a field of view of 17x14cm on 30cm distance
- Mini HDMI to USB-A cable for Basler Camera
- RS-232 to USB-A cable for Zaber Linear Stage
- Mounting components
- A printed A4 sheet with a QR code on it
- 30cm ruler

A.3.2 Experiment Setup

A top-down overview of the setup of the first experiment is shown in Figure A.3. The A4 sheet with the QR code is mounted on the carriage of the linear stage. The camera is setup at a 300mm distance from the measured surface with the camera orthogonal to the starting position of the measured surface and mounted at the same height as the vertical middle of the A4 sheet such that the camera is aiming directly at the middle of the sheet.

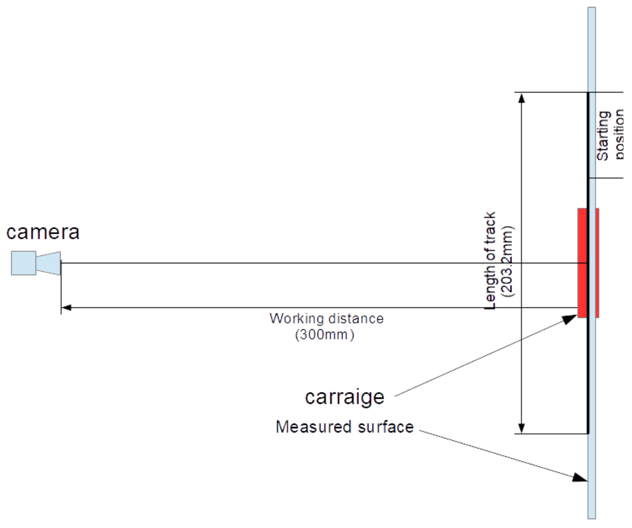


Figure A.3 Setup overview of the first experiment

A different setup was used for the second experiment. A sideview of this setup is shown in Figure A.4. Using this setup, the camera is aimed downwards at the measured surface, which lays flat. The camera is mounted on the motorized linear stage so that it can be moved with a controlled velocity.

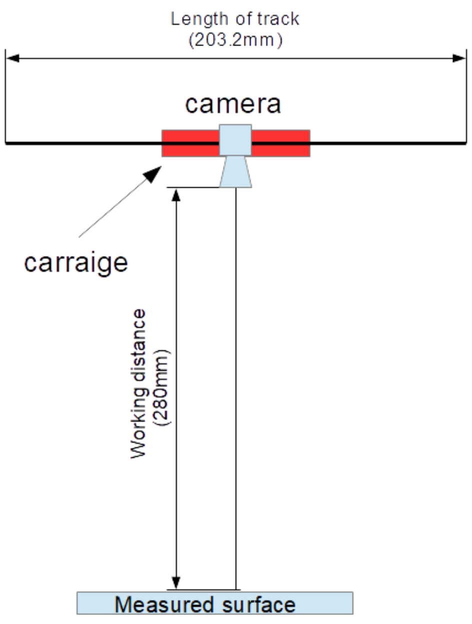


Figure A.4 Sideview of the setup for the second experiment.

A.3.3 Execution

A.3.3.1 Experiment 1

During the first experiment, the linear stage moves the measured surface, a printed image of a QR code, as shown in Figure A.3 with three different static velocities of 1 mm/s, 5 mm/s, and 10 mm/s in the horizontal direction. For each velocity, the camera is capturing 40 images of the measured surface per second, resulting in 298 frames with a resolution of 1936 by 1216 pixels. Using a ruler shown in the image sequence, the displacement in pixels/frame can be computed, which is then used to compute the reference velocity.

The resulting image sequences are then loaded into a MATLAB script where the optical flow algorithms are used to compute the velocity for each frame sequence in pixels/frame, while the known velocity in pixels/frame is used as a reference velocity for computing the error. Before the frame is used as input for the optical flow algorithms, it is cropped to a size of 200 by 200 pixels, such that only the measured surface is shown on the cropped frame. Computing the velocity of the measured surface based on the motion field is done by taking the average of the horizontal component and the average of the vertical component all non-zero components of the motion field in order to filter out elements which are falsely perceived as non-moving. This results in a separate horizontal velocity and vertical velocity.

The errors for each optical flow algorithm are then computed by subtracting the computed velocity from the reference velocity in pixels/frame, as given by Equation 31. As only a horizontal motion is used, only the horizontal components of the motion are used.

$$error(t) = Vref(t) - Vmeas(t) \tag{31}$$

This error metric is first used to configure the parameters for each optical flow algorithm, minimizing the error for each as much as possible for each velocity. The error values that result from the configured algorithms are then used for analysis.

A.3.3.2 Experiment 2

During the second experiment, image sequences are synthesized by using motion data gathered during a previous research conducted by A. Chizari, T. Knop, B. Sirmacek, F. van der Heijden, and W. Steenbergen (Chizari, Knop, Sirmacek, van der Heijden, & Steenbergen, 2020) where an electromagnetic tracker and a gyroscope were used to measure the rotational and translational velocity of a handheld device. From this data a mean on-surface translation of 9 mm/s, a mean translational speed of 6 mm/s and a maximum measured on-surface speed of 17 mm/s and a maximum translational speed of 12 mm/s are derived.

During the experiment, 10 measurements were conducted. The data resulting from these measurements include on-surface displacement over time, which is used to synthesize artificial frame sequences as the on-surface point can be seen as the point at which the camera points.

The velocity can then be estimated from the on-surface location as the derivative of the location, which is the new location minus the old location. This velocity data can then be used as a reference for velocity calculation based on the different optical flow algorithms.

The frame sequences that are used for this synthesis are two image sequences, where one image sequence contains a backhand skin surface, and the other image sequence contains a printed image of a QR code. To keep the backhand skin surface as stationary as possible, the hand is placed flat on a horizontal surface, while the camera moves over the surface, while being aimed downwards as shown in Figure A.4.

To synthesize the frame sequences as accurately as possible, it is important to keep the resolution of the displacement in pixels/frame as high as possible so that small motions can be synthesized as well. Therefore, during the image acquisition, the linear stage moves the camera with a velocity of 0.5 mm/s, while the camera captures 98.5 frames per second, capturing 5000 frames of size 798 by 832 pixels. Using a ruler shown in the image sequence, the displacement in pixels/frame can be computed, which is then used to transform the reference velocity in mm/s to pixels/frame.

Using the two acquired image sequences, 20 image sequences were synthesized using the 10 motion data sets, resulting in 10 image sequences where the QR code is the measured surface, and 10 image sequences where the backhand is the measured surface. These image sequences were then used as input for the analyzed optical flow algorithms, which were optimized during the first experiment, except the optical flow algorithm which was previously implemented for motion measurement. Before the frame is used as input for the optical flow algorithms, it is cropped to a size of 200 by 200 pixels, such that only the measured surface is shown on the cropped frame. The resulting flows were converted to a translational motion vector using the same method as the first experiment. The resulting velocity is then subtracted from the reference velocity, resulting in the error over time for each algorithm.

A.4 Results

The first experiment results in three sets of error values over time for each of the four algorithms. From these error values, the mean absolute error and standard deviation in pixels/frame were calculated, which are shown in Table A.1, Table A.2, and Table A.3.

1 mm/s	Horn-Schunck	Lukas-Kanade	Lukas-Kanade DoG	Farnebäck
Mean absolute error (px/frame)	0.0750	0.1397	0.0231	0.0295
Error standard deviation (px/frame)	0.2565	0.1258	0.1235	0.0470

Table A.1 Error statistics resulting from the optical flow algorithms when applied to a surface with constant velocity of 1 mm/s.

5 mm/s	Horn-Schunck	Lukas-Kanade	Lukas-Kanade DoG	Farnebäck
Mean absolute error (px/frame)	0.4014	0.7919	0.0540	0.0588
Error standard deviation (px/frame)	0.2382	0.0903	0.1753	0.2045

Table A.2 Error statistics resulting from the optical flow algorithms when applied to a surface with constant velocity of 5 mm/s.

10 mm/s	Horn-Schunck	Lukas-Kanade	Lukas-Kanade DoG	Farnebäck
Mean absolute error (px/frame)	0.7337	1.9048	0.1200	0.1186
Error standard deviation (px/frame)	0.3866	0.0641	0.3227	0.4388

Table A.3 Error statistics resulting from the optical flow algorithms when applied to a surface with constant velocity of 10 mm/s.

The second experiment resulted in 20 sets of error values over time, two for each used motion dataset: one for the QR code surface, and one for the backhand surface. The error values were used to compute the mean absolute error in px/frame and in percentage, as an error percentage can be compared to a maximum tolerable percentage. The standard deviations of the error values were computed as well. These metrics were computed over all the datasets per measured surface, resulting in one value for each metric per surface. The resulting values are given in Table A.4 and Table A.5.

QR	Lukas-Kanade DoG	Farnebäck	Affine
Mean absolute error (px/frame)	2.2877	0.1227	2.5571
Error standard deviation (px/frame)	3.4184	1.9837	4.2040
Mean absolute error (%)	562.5246	7.2574	400.8740
Error standard deviation (%)	9.3584e+03	30.2253	2.2036e+03

Table A.4 Error statistics from the optical flow algorithms when applied to QR code surface frame sequences with realistic motion.

Backhand skin	Lukas-Kanade DoG	Farnebäck	Affine
Mean absolute error (px/frame)	2.1269	0.6292	4.6311
Error standard deviation (px/frame)	3.2742	2.3042	8.6170
Mean absolute error (%)	413.3012	21.7943	672.6468
Error standard deviation (%)	6.9850e+03	36.3434	2.8270e+03

Table A.5 Error statistics from the optical flow algorithms when applied to backhand surface frame sequences with realistic motion.

Each individual set of error values was also used to produce boxplots of the error statistics. Per surface, two sets of boxplots were generated which compare the performance of the optical flow algorithms on the different motion datasets. For each measured surface, a set of boxplots was generated comparing the error in pixels/frame, and a set of boxplots was generated comparing the error in percentage. Figure A.5 compares the error values in pixels/frame of each algorithm for all datasets when applied to the printed QR code surface. Figure A.6 does this while expressing the error in percentage for the QR code surface. Figure A.7 compares the error values in pixels/frame of each algorithm for all datasets when applied to the backhand surface. Figure A.6 and Figure A.8 do this while expressing the error in percentage for the backhand surface.

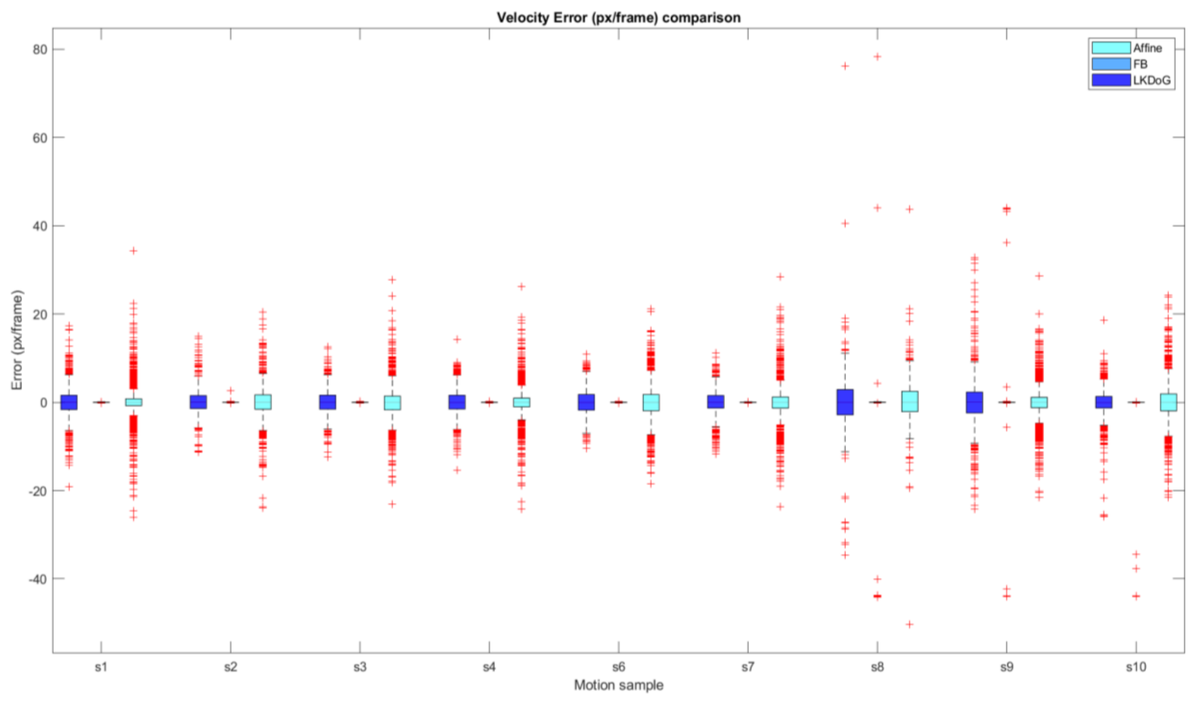


Figure A.5 A comparison of the performance using the error in pixels/frame of each optical flow algorithm on the different motion datasets when applied to the QR code surface.

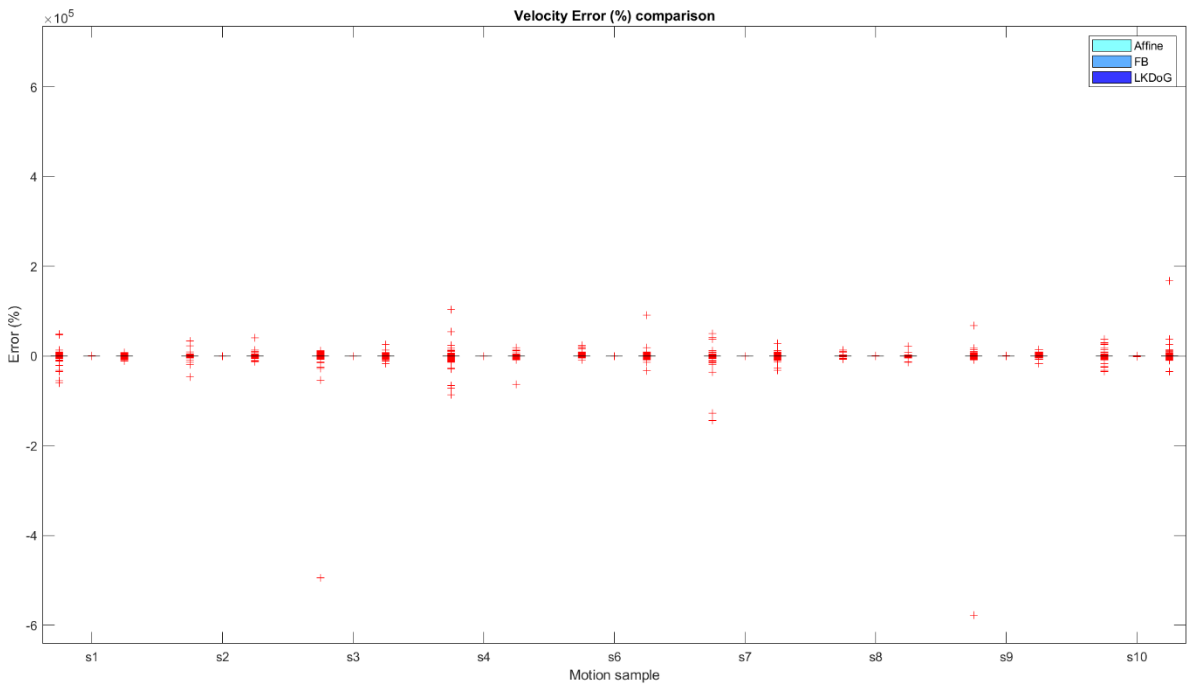


Figure A.6 A comparison of the performance using the error in % of each optical flow algorithm on the different motion datasets when applied to the QR code surface.

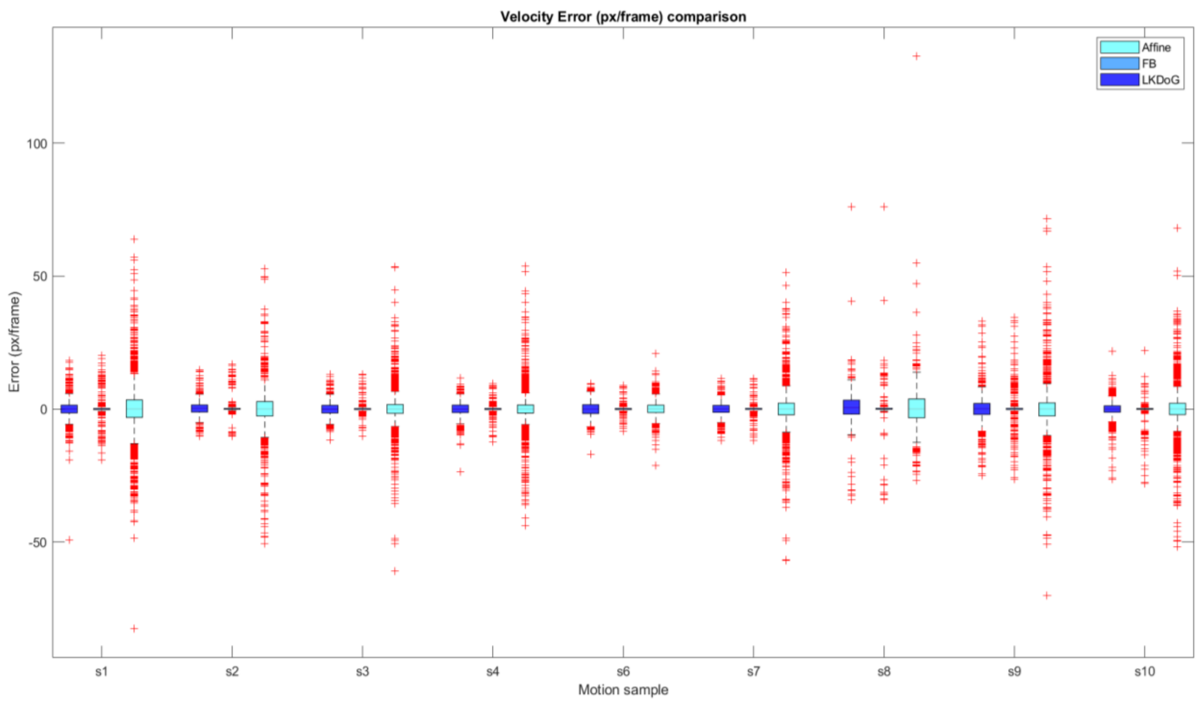


Figure A.7 A comparison of the performance using the error in pixels/frame of each optical flow algorithm on the different motion datasets when applied to the backhand surface.

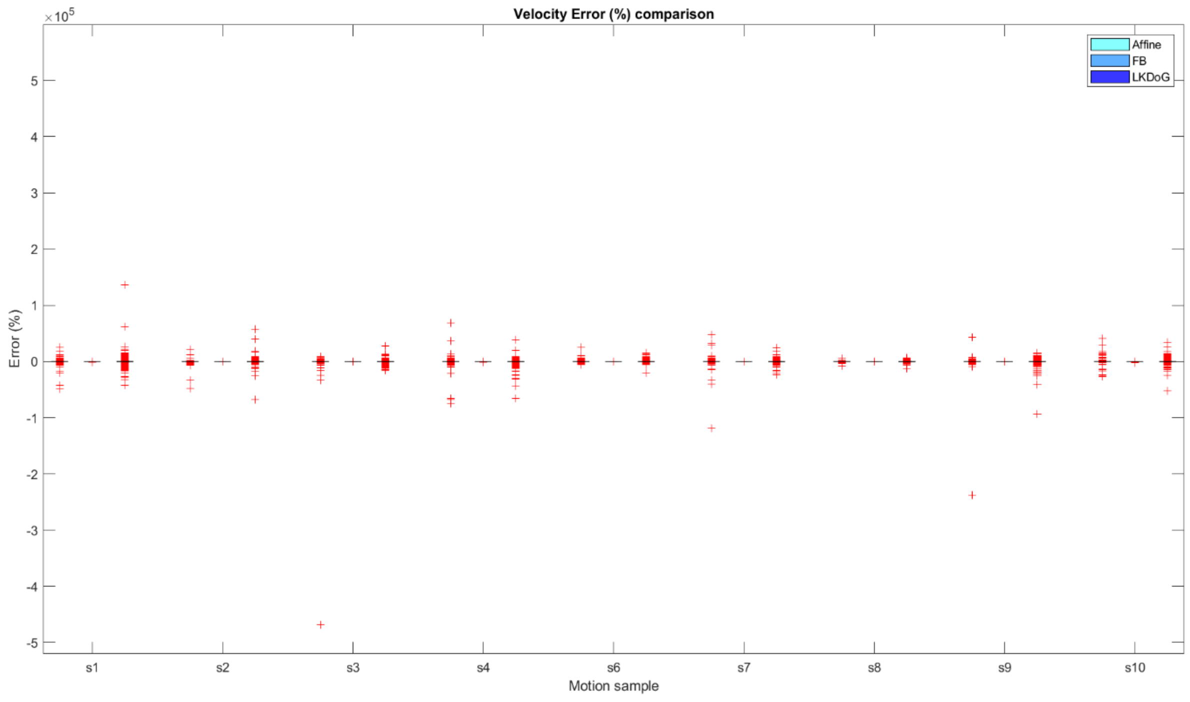


Figure A.8 A comparison of the performance using the error in % of each optical flow algorithm on the different motion datasets when applied to the backhand surface.

Finally, the error is plotted against the reference velocity in order to visualize the relation between the error and the real displacement for each algorithm. This was done by collecting all reference velocities from the different motion datasets and sorting them in ascending order. Then the absolute error in pixels/frame and percentage is given on the vertical axis for each algorithm.

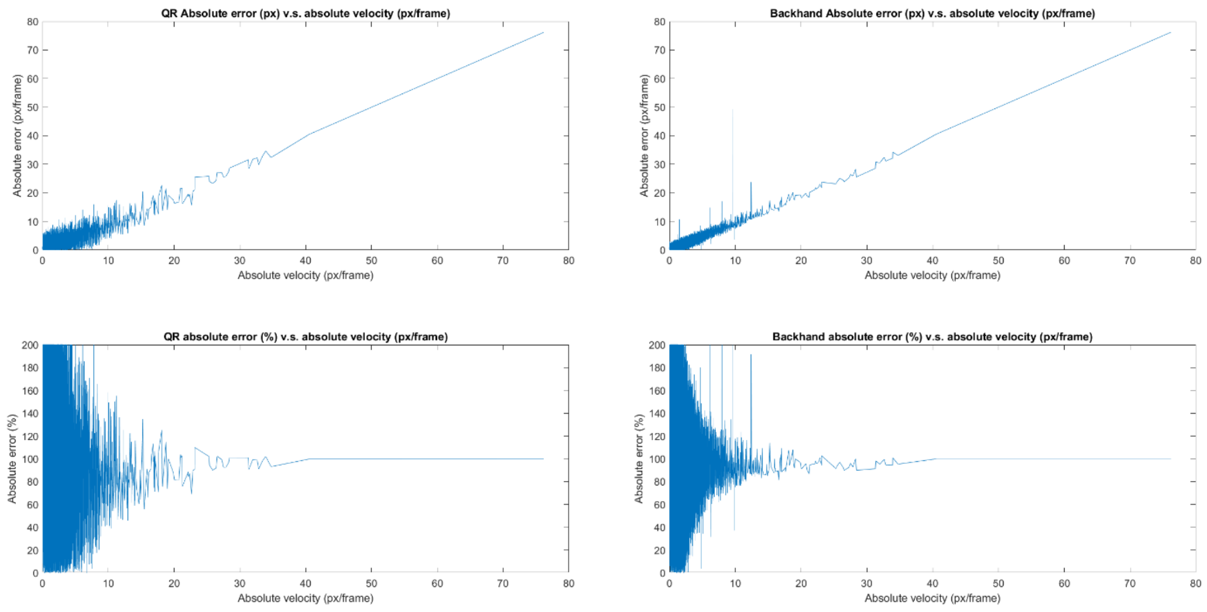


Figure A.9 The error versus real velocity for the Lukas-Kanade with derivative of gaussian algorithm. Top left: error from the QR code surface measurement in pixels/frame. Top right: error from the backhand surface measurement in pixels/frame/ Bottom left: error from the QR code surface measurement in percentage. Bottom right: error from the backhand surface measurement in percentage.

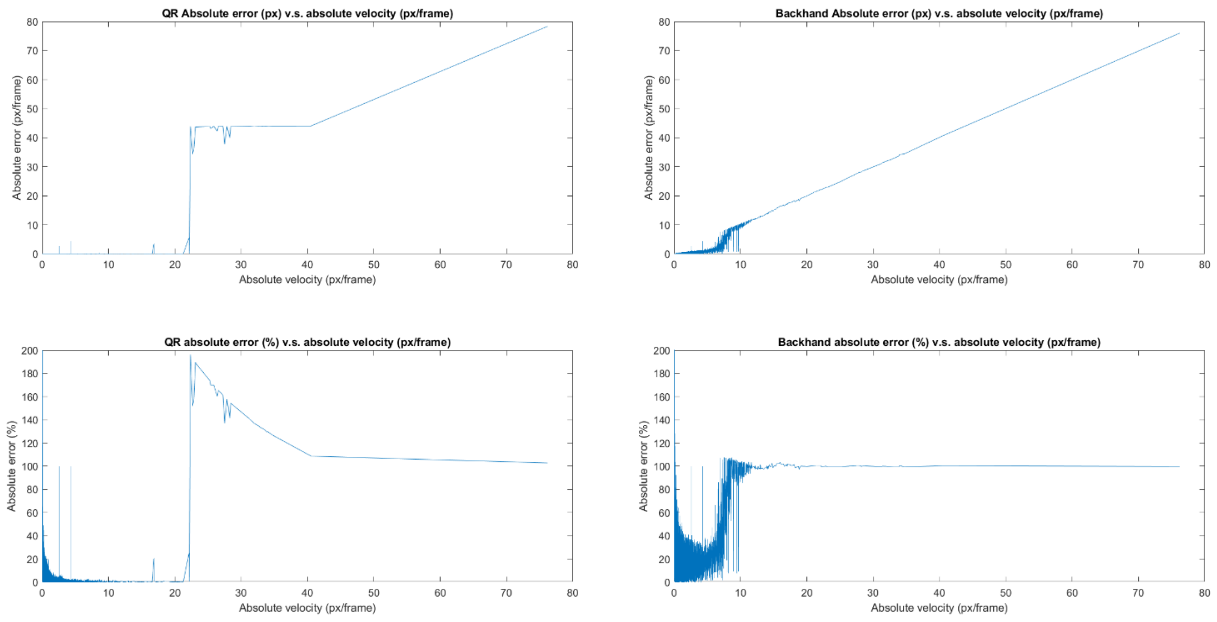


Figure A.10 The error versus real velocity for the Farnebäck algorithm. Top left: error from the QR code surface measurement in pixels/frame. Top right: error from the backhand surface measurement in pixels/frame/ Bottom left: error from the QR code surface measurement in percentage. Bottom right: error from the backhand surface measurement in percentage.

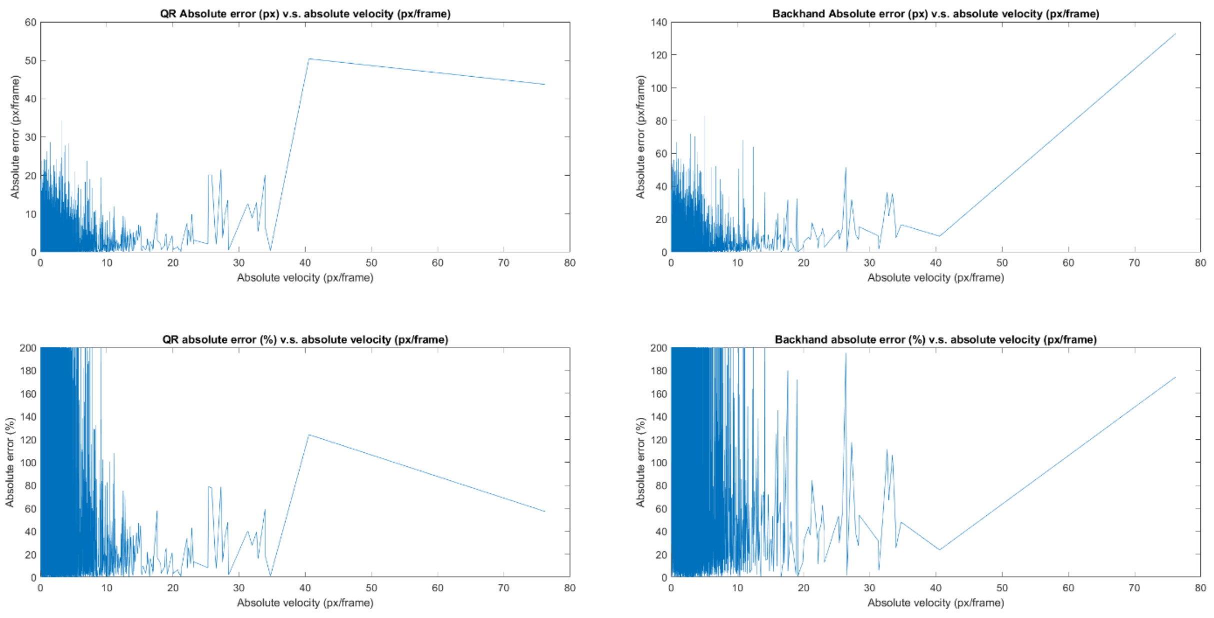


Figure A.11 The error versus real velocity for the Affine flow algorithm. Top left: error from the QR code surface measurement in pixels/frame. Top right: error from the backhand surface measurement in pixels/frame/ Bottom left: error from the QR code surface measurement in percentage. Bottom right: error from the backhand surface measurement in percentage.

Please note that the density of real velocities is not even and therefore gaps exist between the highest velocity and the second highest velocity. This makes sections of the curve seem linear, while this does not have to be true.

A.5 Discussion

The first experiment provides means of comparing the four different optical flow algorithms using their error. From Table A.1 to Table A.3 it can be seen that for all optical flow algorithms, the mean error increases when the velocity increases. It is also clear that with a maximum mean error of 0.1200 and 0.1186 pixels/frame, the Lukas-Kanade with derivative of gaussian and the Farnebäck algorithms are the best performing algorithms as they manage to reconstruct the velocity the most accurately when the velocity is static. Therefore, these two algorithms were chosen for further analysis during the next experiment. While their mean error values are very close on all measured velocities, the Farnebäck algorithm has a higher standard deviation when the velocity is 5 mm/s and 10 mm/s, meaning that it performs less consistent than the Lukas-Kanade with derivative of gaussian algorithm on higher velocities.

The second experiment shows that for a realistic motion, the Farnebäck algorithm performs better than the Lukas-Kanade with derivative of gaussian algorithm and the previously implemented affine flow algorithm. When executed on both the QR code surface measurement and the backhand skin measurement, the Farnebäck algorithm has the lowest mean absolute error in pixels/frame and percentage, which is relative to the real velocity. It also seems to perform more consistent as the standard deviation is lower for the Farnebäck algorithm when compared to the Lukas-Kanade and affine flow algorithms. This is also shown by the boxplot sets shown in Figure A.5 to Figure A.8 for both surfaces. The errors for the Farnebäck algorithm have minima and maxima and first and third quartiles which lay closer to the median in all cases than the other two analyzed algorithms.

It is noteworthy that the performance difference for the Farnebäck and affine flow algorithms is rather large between measured surfaces. This is assumed to be because the skin surface has a lower contrast than the QR code surface, where local optima can be more easily found by the algorithms, solving the optical flow Equation. This does, however, not seem to be the case for the Lukas-Kanade with derivative of gaussian algorithm as it performs better on the backhand surface than on the QR code surface.

As was observed during the first experiment, absolute error values seem to increase when the real velocity increases. Figure A.9 shows that this is clearly the case of the Lukas-Kanade with derivative of gaussian algorithm, where there seems to be a linear relationship where the error in pixels/frame increases as the velocity increases. It can also be observed that for smaller velocities, the relative error varies highly, while when the real velocity is higher than 25 pixels/frame, the measured velocity seems to be consistently double that of the real velocity as the error becomes 100% for both the backhand surface and the QR surface.

Figure A.10 shows that a similar relationship seems to exist for the Farnebäck algorithm for the backhand surface measurement when the real velocity is over 10 pixels/frame. However, the error seems to be increasing much slower when the real velocity is under 7 pixels/frame, making for a more reliable measurement. This conclusion cannot be drawn when looking at the performance of the Farnebäck algorithm on the QR code surface. On this surface, the algorithm has a very low error for displacements under 20 pixels/frame. The error between a velocity of 20 pixels/frame and 40 pixels/frame seems to be constant. Please note that the velocity density is not even, and larger gaps exist between higher velocities. Therefore, nothing can be said about the linearity over velocities with big gaps. However, as can be seen when looking at the error in percentage, the error relative to the real velocity becomes higher for very low velocities. This is assumed to happen when the displacement is less than one pixel/frame as the image sequence then shows almost no displacement and therefore almost no optical flow. Nonetheless, it is recommended when using the Lukas-Kanade with derivative of gaussian and Farnebäck algorithms for motion measurement for handheld perfusion imaging to keep the maximum observed displacement under 7 pixels/frame for a consistently low error in pixels/frame. This can be done by increasing the frequency of the image acquisition, as the velocity per period becomes smaller.

For the affine flow algorithm, the only relationship between the error and real velocity that seems from Figure A.11, is that the error is very inconsistent and could increase when the real velocity decreases and increases when the velocity is higher than 20 pixels/frame. There seems to be a point where the algorithm performs relatively well, which is around 20 pixels/frame. This could be due to wrong configuration of the parameters of the algorithm.

A.6 Conclusion

During this research, the Horn-Schunk, Lukas-Kanade, Lukas-Kanade with derivative of gaussian, and Farnebäck algorithms for optical flow computation were analyzed and compared to each other and with a previously implemented algorithm in order to select the best performing algorithm for motion measurement for handheld perfusion imaging. A theoretical analysis of the four algorithms was performed to gain a better understanding of their implementation in software and to be used as a guidance when configuring the algorithm parameters. The algorithms were then evaluated during two experiments. The goal of the first experiment was to select the best performing optical flow algorithms by providing frame sequences with a surface that moves with various constant velocities and comparing the errors resulting from the velocity estimates. The Lukas-Kanade with derivative of gaussian and Farnebäck algorithms seemed to perform the best and were evaluated during the second experiment, where the algorithms were compared against the previously implemented affine flow algorithm, which is based on the Lukas-Kanade algorithm. This experiment used image sequences with realistic motion introduced by synthesizing image sequences of a backhand skin surface and a printed QR code surface using motion data gathered during previous research. Comparison between the algorithms was based on the absolute error values over time in pixels/frame and the relative error values, expressed in percentage. From this analysis it was shown that the Farnebäck optical flow algorithm performs best on realistic frame sequences when used for handheld perfusion imaging and it was recommended to keep the measured displacement underneath 7 pixels/frame.

B Appendix 2: The Graphical Processing Unit (GPU)

The Graphical Processing Unit (GPU) is a computer architecture consisting of many parallel floating-point units and utilizes thread-level parallelism, data-level parallelism, and instruction-level parallelism to highly accelerate the execution of parallelized algorithms (Hennessy & Patterson, 2012). The GPU exploits data-level parallelism by parallelly applying a single instruction to a collection of data using streaming multiprocessors, which manages, schedules, and executes 32 threads (a warp). This architecture is called the Single-Instruction, Multiple-Thread (SIMT) architecture (NVIDIA Corporation, 2021). A SIMT processor is pipelined such that it can exploit instruction-level parallelism and SIMT processors in a GPU are multithreaded to exploit thread-level parallelism.

GPUs are usually combined with a Central Processing Unit (CPU) to create a heterogeneous system where the CPU is called the host and the GPU is called the device. Both processing units can then be utilized to perform parallel computation.

B.1 Compute Unified Device Architecture (CUDA)

One of the most used brands of GPUs is NVIDIA, which provide the Compute Unified Device Architecture (CUDA) programming model (NVIDIA Corporation, 2021). CUDA enables developers to program CUDA compatible GPUs using C++ and provides a runtime library to interface the device memory, call device specific functions, and manage systems with multiple GPU's.

The CUDA model defines a thread hierarchy and an accompanying memory hierarchy. Threads are executed in groups of 32 on a SIMT processor. This collection of threads is called a warp. A maximum of 1024 threads in CUDA make up a 1-, 2-, or 3-dimensional block, called a thread block. Multiple thread blocks can be defined to form a grid, which can also be 1-, 2-, or 3-dimensional. This way, a CUDA kernel can be executed on over 1024 threads in parallel. A structure like this simplifies working with multidimensional data structures such as 2D or 3D matrices.

CUDA threads have access to multiple memory spaces on off-chip DRAM, which is separate from the host memory. Each thread in a block has access to its own private memory, which is a part of off-chip DRAM and threads within a block can communicate with each other through shared memory. All threads have access to the same global memory, which is the largest chunk of device memory and can be accessed from the host processor.

B.2 The Tensor core

The NVIDIA Tensor core was first introduced as a part of the NVIDIA Volta Architecture and is a processor capable of performing the operation $D = A \times B + C$ on a 4 by 4 matrix in one clock cycle (NVIDIA Corporation, 2017). Here matrices A and B are of 16-bit precision and matrices C and D are either FP16 or FP32 matrices. This processor core greatly reduces the amount of clock cycles needed to perform such matrix operations, which results in a major speedup of matrix-matrix multiplications. Support for higher precision matrix operations on Tensor cores was added on the NVIDIA Turing architecture, where support for INT4 and INT8 precision was added, which doubles or quadruples the execution rate respectively (Corporation, 2018).

Bibliography

Alcantarilla, P. F., Bartoli, A., & Davidson, A. J. (2012, October). KAZE Features. 214-227. doi:10.1007/978-3-642-33783-3_16

Barron, J., Fleet, D., & Beauchemin, S. (1994). Performance of Optical Flow Techniques. *International Journal of Computer Vision, 12*(1), 43-77.

Chizari, A., Knop, T., Sirmacek, B., van der Heijden, F., & Steenbergen, W. (2020, April 3). Exploration of Movement Artefacts in Handheld Laser Speckle Contrast Perfusion Imaging. *Biomedical Optics Express*, pp. 2352-2365.

Corporation. (2018). *Nvidia Turing GPU Architecture.* Retrieved Februari 16, 2021, from https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

de Souza Rosa, L., Dasu, A., Diniz, P. C., & Bonato, V. (2018). A Faddeev Systolic Array for EKF-SLAM and its Arithmetic Data Representation Impact on FPGA. *Jornal of Signal Processing Systems, 90*(3), 357-369.

Escobar, F. A., Chang, X., & Valderrama, C. (2016). Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC. *IEEE Transactions on Parallel and Distributed Systems, 27*(2), 600-612.

Farnebäck, G. (2003). Two-Frame Motion Estimation Based on Polynomial Expansion. *Image analysis.* Halmstad.

Giubilato, R., Chiodini, S., Pertile, M., & Debei, S. (2019). An evaluation of ROS-compatible stereo visual SLAM methods on a nVidia Jetson TX2. *Measurement, 140*, 161-170.

Grimm, S. (2020). Design of a 3D imaging system for psoriasis assesment. Enschede, Overijssel, The Netherlands: University of Twente.

Hartley, R., & Zisserman, A. (2004). Camera Models. In *Multiple View Geometry* (pp. 153-157). New York: Cambridge University Press.

Hennessy, J. L., & Patterson, D. A. (2012). *Computer Architecture A Quantitative Approach.* Waltham: Morgan Kaufmann.

Horn, B. K., & Schunck, B. G. (1981). Determining Optical Flow. *Artificial Intelligence, 17*(1), 185-203.

Idris, M. Y., Arof, H., Noor, N. M., Tamil, E. M., & Razak, Z. (2012). A co-processor design to accelerate sequential monocular SLAM EKF process. *Measurement, 45*(8), 2141-2152.

Jiménez Serrata, A. A., Yang, S., & Li, R. (2017). An intelligible implementation of FastSLAM2.0 on a low-power embedded architecture. *EURASIP Journal on Embedded Systems, 2017*(27).

Jones, D. H., Powell, A., Bouganis, C.-S., & Cheung, P. Y. (2010). GPU versus FPGA for high productivity computing. *International Conference on Field Programmable Logic and Applications,* (pp. 119-124). Milano, Italy. doi:10.1109/FPL.2010.32

Karmakar, R., Chattopadhyay, S., & Chakraborty, S. (2017). Impact of IEEE 802.11n/ac PHY/MAC High Throughput Enhancements on Transport and Application Protocols—A Survey. *IEEE Communications Surveys & Tutorials, 19*(4), 2050.

Khorov, E., Luakhov, A., Kiryanov, A., & Bianchi, G. (2018). A Tutorial on IEEE 802.11ax High Efficiency WLANs. *IEEE Communications Surveys & Tutorials, 21*(1), 197-216.

Kohlhoff, C. (2021, 03 29). *Boost.Asio*. Retrieved from Boost: https://www.boost.org/doc/libs/1_73_0/doc/html/boost_asio.html

Li, J., Deng, G., Zhang, W., Zhang, C., Wang, F., & Liu, Y. (2020). Realization of CUDA-based real-time multi-camera visual SLAM in embedded systems. *Journal of Real-Time Image Processing, 17*(3), 713-727.

Minhas, U. I., Bayliss, S., & Constantinides, G. A. (2014). GPU vs FPGA: A comparative analysis for non-standard precision. In S. M. Goehringer D., *Reconfigurable Computing: Architectures, Tools, and Applications* (Vol. 8405, pp. 298–305). Springer, Cham. doi:10.1007/978-3-319-05960-0_32

Montemerlo, M., & Thrun, S. (2007). *FastSLAM.* Berlin: Springer-Verlag.

Mukunoki, D., Ozaki, K., Ogita, T., & Imamura, T. (2020). DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions. In C. B. Sadayappan P., *High Performance Computing. ISC High Performance 2020. Lecture Notes in Computer Science* (Vol. 12151, pp. 230-248). Springer, Cham. doi:https://doi.org/10.1007/978-3-030-50743-5_12

Nurvitadhi, E., Venkatesh, F., Sim, J., Marr, S., Huang, R., Gee Hock Ong, J., . . . Boudoukh, G. (2017). Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* Monterey California USA.

NVIDIA Corporation. (2017, August). *Nvidia Tesla V100 GPU Architecture.* Retrieved Februari 16, 2021, from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

NVIDIA Corporation. (2021, Februari 9). *CUDA C++ Programming Guide.* Retrieved Februari 9, 2021, from https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Object for estimating optical flow using Farneback method. (n.d.). (Mathworks) Retrieved 06 02, 2020, from https://mathworks.com/help/vision/ref/opticalflowfarneback.html

Object for estimating optical flow using Horn-Schunck method. (n.d.). (Mathworks) Retrieved 05 25, 2020, from https://mathworks.com/help/vision/ref/opticalflowhs.html

Object for estimating optical flow using Lucas-Kanade derivative of Gaussian method. (n.d.). (Mathworks) Retrieved 05 27, 2020, from https://mathworks.com/help/vision/ref/opticalflowlkdog.html

Object for estimating optical flow using Lucas-Kanade method. (n.d.). (Mathworks) Retrieved 05 26, 2020, from https://mathworks.com/help/vision/ref/opticalflowlk.html

Oshana, R. (2013). Chapter 11 - Optimizing Embedded Software for Performance. In R. Oshana, & M. Kraeling, *Software Engineering for Embedded Systems* (pp. 313-342). Elsevier.

Paz, L. M., Piniés, P., Tardós, J. D., & Neira, J. (2008). Large-Scale 6-DOF SLAM With Stereo-in-Hand. *IEEE Transactions on Robotics, 24*(5), 946-957.

Piat, J., Fillatreau, P., Tortei, D., Brenot, F., & Devy, M. (2020). HW/SW co-design of a visual SLAM application. *Journal of Real-Time Image Processing, 17*(3), 667-689.

Shah, N. (2020). 3D Stereovision for quantification of Skin Diseases. Enschede, Overijssel, The Netherlands: University of Twente, Control Laboratory, EL/RAM, Faculty of Electrical Engineering, Mathematics & Computer Science.

Shanker, S., & Shameer, M. (2016). Time Complexity of Matrix Transpose Algorithm using Identity Matrix as Reference Matrix. *International Journal of Computer Science and Information Technologies*, 2347-2348.

Sharma, G., Agarwala, A., & Bhattacharya, B. (2013). A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. *Computers and Structures*, 31-37.

Sharma, P., Vleugels, R., & Nambudiri, V. E. (2019). Augmented reality in dermatology: Are we ready for AR? *Journal of the American Academy of Dermatology, 81*(5), 1216-1222.

Skiena, S. S. (2008). Matrix Multiplication. In *The Algorithm Design Manual* (pp. 45-46). London: Springer-Verlag.

Steenbergen, W. (2020). *Progress report.* University of Twente, Enschede.

Tertei, D. T., Piat, J., & Devy, M. (2016). FPGA design of EKF block accelerator for 3D visual SLAM. *Computers and Electrical Engineering, 55*, 123-137.

Teubner, J., & Woods, L. (2013). *Data Processing on FPGAs.* Morgan & Claypool.

Thrun, S., Burgard, W., & Fox, D. (2006). *Probabilistic Robotics.* Cambridge, Massachusetts: MIT Press.

Thrun, S., Liu, Y., Koller, D., Ng, A. Y., Ghahramani, Z., & Durrant-Whyte, H. (2004). Simultaneous Localization and Mapping with Sparse Extended Information Filters. *The International Journal of Robotics Research, 23*(7-8), 693-716.

van der Heijden, F. (2016). 3D measurements from camera images.

van der Heijden, F. (2018). Visual Navigation.