applied sciences

MDPI

Article

Context-Adaptable Deployment of FastSLAM 2.0 on Graphic Processing Unit with Unknown Data Association

Jessica Giovagnola [1,2], Manuel Pegalajar Cuéllar [3,*] and Diego Pedro Morales Santos [2]

[1] Infineon Technologies AG, Am Campeon 1-15, 85579 Neubiberg, Germany; jessica.giovagnola@infineon.com
[2] Department of Electronic and Computer Technology, University of Granada, Avenida de Fuente Nueva s/n, 18071 Granada, Spain; diegopm@ugr.es
[3] Department of Computer Science and Artificial Intelligence, University of Granada, Calle Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain
* Correspondence: manupc@decsai.ugr.es

Abstract: Simultaneous Localization and Mapping (SLAM) algorithms are crucial for enabling agents to estimate their position in unknown environments. In autonomous navigation systems, these algorithms need to operate in real-time on devices with limited resources, emphasizing the importance of reducing complexity and ensuring efficient performance. While SLAM solutions aim at ensuring accurate and timely localization and mapping, one of their main limitations is their computational complexity. In this scenario, particle filter-based approaches such as FastSLAM 2.0 can significantly benefit from parallel programming due to their modular construction. The parallelization process involves identifying the parameters affecting the computational complexity in order to distribute the computation among single multiprocessors as efficiently as possible. However, the computational complexity of methodologies such as FastSLAM 2.0 can depend on multiple parameters whose values may, in turn, depend on each specific use case scenario (ingi.e., the context), leading to multiple possible parallelization designs. Furthermore, the features of the hardware architecture in use can significantly influence the performance in terms of latency. Therefore, the selection of the optimal parallelization modality still needs to be empirically determined. This may involve redesigning the parallel algorithm depending on the context and the hardware architecture. In this paper, we propose a CUDA-based adaptable design for FastSLAM 2.0 on GPU, in combination with an evaluation methodology that enables the assessment of the optimal parallelization modality based on the context and the hardware architecture without the need for the creation of separate designs. The proposed implementation includes the parallelization of all the functional blocks of the FastSLAM 2.0 pipeline. Additionally, we contribute a parallelized design of the data association step through the Joint Compatibility Branch and Bound (JCBB) method. Multiple resampling algorithms are also included to accommodate the needs of a wide variety of navigation scenarios.

Keywords: FastSLAM2.0; CUDA; GPGPU; JCBB

Citation: Giovagnola, J.; Cuéllar, M.P.; Santos, D.P.M. Context-Adaptable Deployment of FastSLAM 2.0 on Graphic Processing Unit with Unknown Data Association. *Appl. Sci.* **2024**, *14*, 11466. https://doi.org/10.3390/app142311466

Academic Editor: André Sales Mendes

Received: 23 October 2024
Revised: 22 November 2024
Accepted: 6 December 2024
Published: 9 December 2024

1. Introduction

Simultaneous Localization and Mapping (SLAM) plays a vital role in the implementation of autonomous navigation systems, as it enables an agent to estimate its current pose without previous knowledge of its surrounding environment. In more detail, SLAM estimates the robot's position while incrementally building an environmental representation (i.e., a map). Its rise in popularity has been strictly concurrent with the growing variety of applications for autonomous systems, which include, but are not limited to, autonomous cars [1], unmanned aerial vehicles, commonly known as drones [2], underwater robotics [3], space exploration [4], and indoor navigation [5–7]. The localization and mapping process leverages a combination of one or more sensors [8,9], which can be classified as *exteroceptive*, i.e., aimed at perceiving the surrounding environment (e.g., camera, LiDAR), and *proprioceptive*, which can sense the position, orientation, and movement of the agent in space (e.g.,

IMU, encoder, GPS). The fusion of one or more sensing modalities is made necessary by the impossibility of leveraging GPS technology as a stand-alone sensing technique under several use cases [10] since autonomous systems need high-level positioning accuracy and consistent signal. However, despite the advent of high precision GNSS systems (e.g., Real-Time Kinematics—RTK), centimeter-level positioning accuracy is guaranteed only under specific conditions, suffering loss of signal under unfavorable atmospheric conditions or infrastructural characteristics (e.g., indoors, presence of high buildings causing canyoning effects) [11]. The sensor information is consequently elaborated through an algorithm pipeline that commonly approaches the SLAM problem in a probabilistic fashion. The current state-of-the-art SLAM methodologies can be grouped into two major categories, namely, *filter-based methods* and *optimization-based methods* [12,13].

Filter -based methods can be considered direct derivations of Bayesian filtering [14], as they consist of two-step iterative processes, namely, the following:

- *The prediction step*, where the new state is estimated according to an *evolution model* and a control input;
- *The correction step*, where the current observations are matched against the previously observed features.

Between the prediction and the correction step, the acquired observations must be associated with the already existing partial environmental reconstruction. If one or more observations are not matched with any of the pre-existing map features, new features are initialized. Some SLAM proposals model the map features as uniquely identifiable (*known data association*). In this case, the data association problem is addressed deterministically. However, a probabilistic approach is needed whenever the map features are not uniquely identifiable (*unknown data association*). In this dissertation, we address the unknown data association case for higher generality.

The earliest SLAM proposals belong to the filter-based category and are based on the Kalman Filter (KF) [15], such as the Extended Kalman Filter (EKF) [16] and its variations, among which we also mention the Unscented Kalman Filter (UKF) [17] and the Information Filter (IF) [18]. The main limitations of such approaches are related to handling nonlinearities, the Gaussian modeling of uncertainty, and the increasing dimension of the state as long as new features are observed. These problems are tackled in Particle Filter (PF)-based methods [19] such as FastSLAM 1.0 [20] and FastSLAM 2.0 [21], where a proposal distribution is sampled through a set of weighted hypotheses, called *particles*.The weight of each particle is proportional to the correspondence between the prediction it holds and the observed state values. In filter-based methods, the map can be represented as an Occupancy Grid or a Landmark-Based Map [22]. Occupancy Grids model the environment as an array of cells, i.e., uniformly dimensional portions of space (squares or cubes). Each cell holds the probability of being free or occupied. On the other hand, Landmark-Based Maps consist of a set of points, commonly called *landmarks*, whose position is fixed in the space. The robot's pose estimation is based on the measurement of its distance from the observed landmarks. In this case, each landmark l is identified with a 3D vector $[x_l, y_l, l]$ containing its coordinates with respect to the map origin $[x_l, y_l]$ and its numerical identifier $l \in [0, N_l)$. While Occupancy Grids have higher information content, Landmark-Based Maps provide a more compact environmental representation, reducing memory consumption [23,24].

Optimization-based methods model the map as factor graphs. They can encode information at a geometric level (e.g., points, lines, planes) and a semantic level (instances of objects stored in a previously defined database).

Further insights on the main state-of-the-art SLAM proposals available are given in review works such as [12,13,25–29]. While filter-based SLAM is more employed for *online SLAM*, where the latest robot pose is estimated jointly with the map, optimization-based SLAM is mainly leveraged to solve the *full SLAM* problem, where the whole trajectory is estimated [12].

This work focuses on particle filter-based algorithms, specifically FastSLAM 2.0, based on Landmark-Based Maps. FastSLAM 2.0 leverages a Rao–Blackwellized particle filter

where the robot's pose and the map estimations are decoupled. In more detail, the robot pose estimation is first predicted according to the motion model and the control inputs, then adjusted considering the latest associated observations. On the other hand, each landmark position is estimated using an extended Kalman Filter. Further details on FastSLAM 2.0 are discussed in Section 2.

One of the main limitations of SLAM approaches is their high computational complexity, and particle filter-based methods are no exception. In addition, SLAM should often run on onboard devices and guarantee real-time pose estimation. In this context, parallel programming and devices enabling massive parallelization, such as Graphic Processing Units (GPUs), can enable many SLAM methodologies to meet real-time constraints [30–34]. PF-based approaches can particularly benefit from acceleration via parallel programming due to their modular structure. In fact, particles are calculated independently and, within each particle, the landmark poses are also independently estimated. In this context, the most commonly adopted acceleration modality in the literature is particle-wise parallelization, as in [35–39]. Some proposals focus the acceleration efforts on specific steps of the pipeline, such as data association [40,41], resampling [42–44], and particle weight computation [37]. As for FastSLAM 2.0, a few works provide parallelized implementations on the edge. An accelerated implementation for FPGA is available in [45], while a parallelized implementation of the algorithm for the OMAP4430 architecture is available at [46]. To the best of our knowledge, the only parallelized implementation of FastSLAM 2.0 on a GPU device is discussed in [39], where a particle-wise parallelization for monocular inertial SLAM is proposed. The FAST Corner Detection method carries out the data association step on the CPU.

The parallelization process requires identifying the parameters that affect the algorithm's computational complexity to distribute the computation among the single multiprocessors efficiently. In some cases, such as in FastSLAM 2.0, the complexity can be determined by multiple parameters, which may depend on the use case scenario of interest (i.e., *the context*), providing multiple possibilities in the parallelization design. In addition, the specific characteristics of the hardware architecture in use significantly impact the performance in terms of latency. However, the state-of-the-art does not include any guidelines on addressing parallelization modality selection, but only provides single implementations specifically designed for a single architecture. Consequently, the optimal parallelization modality still needs to be empirically determined, involving the potential need for multiple re-designs. Therefore, in this paper, we propose a context-adaptable design of FastSLAM 2.0 for GPU based on the CUDA programming paradigm. The proposed design enables the implementation of different parallelization modalities depending on the context. By *context*, we mean the values assumed by the set of parameters determining the algorithm's complexity, which, in turn, depends on the use case of interest. Combined with the flexible design for FastSLAM 2.0, we provide an evaluation methodology that enables the assessment of the optimal parallelization modality depending on the context and the hardware architecture in use.

The design proposed in this contribution parallelizes all the functional blocks of FastSLAM 2.0, including the data association step, through the Joint Compatibility Branch and Bound (JCBB) methodology [47]. JCBB enables simultaneous handling of multiple non-uniquely identifiable observations, leveraging a branch-and-bound approach in combination with a joint compatibility test. To the best of our knowledge, this is the first parallelization of FastSLAM 2.0 tackling unknown data association, and no other parallelized version of JCBB is available.

In addition, parallelization of multiple resampling approaches is available with the goal of accommodating the needs of the broadest possible variety of navigation scenarios.

Our methodology was validated by deploying FastSLAM 2.0 onto a General Purpose GPU (GPGPU) interfaced with a simulation environment that included a model of an IMU and a generic range sensor.

The rest of this paper is organized as follows. Section 2 provides theoretical background and discusses the state-of-the-art regarding the data association and the resampling problems. In more detail, Section 2.1 discusses the elements of the FastSLAM 2.0 method, Section 2.2 tackles the data association problem, and the resampling problem is discussed in Section 2.3. Section 3 describes the hardware and software setup: Section 3.1 provides the specification of the hardware, Sections 3.2 and 3.3 summarize the main characteristics of the CUDA programming paradigm and the PyCUDA library, and Section 3.4 addresses the memory access management. Section 4 describes our evaluation methodology, including the simulation environment (Section 4.1), the partitioning of the pipeline into functional blocks (Section 4.2), the analysis of the parameter dependencies (Section 4.3), and the time gain evaluation (Section 4.4). Section 5 tackles the main characteristics of the functional blocks' parallelization. Section 6 is dedicated to the experimental results and discussions thereabout and Section 7 draws the conclusions of this contribution and discusses future developments.

2. Simultaneous Localization and Mapping

This section tackles the theoretical aspects of FastSLAM 2.0 to provide the reader with the necessary background to comprehend the design described in Section 5. In more detail, Section 2.1 and its subsections describe the mathematical details of FastSLAM 2.0's method, Section 2.2 is dedicated to the data association problem, with special focus on the Section Joint Compatibility Branch and Bound methodology, abd Section 2.3 tackles the resampling problem.

SLAM provides self-pose estimation without the need for a preexisting map by fusing information from different sensors, which can either be proprioceptiveor exteroceptive, as described in Section 1. For the sake of our dissertation, we assume the robot is equipped with a range sensor and an inertial sensor (IMU). Consequently, the *prediction model* and the *observation model* are stated in the following sections according to these assumptions.

2.1. FastSLAM 2.0

FastSLAM 2.0, like the other PF-based approaches, provides the robot's and landmarks' pose estimations as discrete samples (particles) of a probabilistic distribution. This characteristic allows for overcoming limitations brought by modeling the robot's pose as Gaussian distributed, as in EKF-based methods [21]. However, the map features are still modeled as Gaussian distributed. The algorithm leverages the conditional independence of map features and the robot's path, as expressed in the following:

$$p(X_t, m \mid z_t) = P(X_t \mid z_t) P(m \mid X_t, z_t)$$

where X_t is the set of the robot poses (x_i, y_i, θ_i), $i = 1, \ldots, t$ from the beginning of the navigation until time t, m is the set of the map features coordinates (called *landmarks*) (x_{lj}, y_{lj}), $j = 1, \ldots, N_l$ (N_l is the number of landmarks), and z_t is the set of observations (ρ_k, β_k), $k = 1, \ldots, N_o$ (N_o is the number of observed features). As described in Section 4.1, the robot is modeled as a rigid body navigating a planar space. Therefore, (x_i, y_i) and (x_{lj}, y_{lj}) are the Cartesian coordinates of the robot and landmark j, expressed in meters, and θ_i is the orientation of the robot, expressed in radians. As for the measurement, for the sake of our dissertation, we assume that the robot is equipped with a generic range sensor. Consequently, the measurements ρ_k and β_k indicate the detections' range (meters) and the bearing (radians). Therefore, the joint distribution of the full path X_t and the map m can be obtained, according to Bayes' Theorem, with the multiplication of the probability of the trajectory X_t given the last measurement z_t by the probability of the map m given the path X_t and the measurement z_t. This consideration allows FastSLAM 2.0 to decouple the map features from the robot poses via a factorization called Rao–Blackwellization, where $P(X_t|z_t)$ is expressed through particles, i.e., a set of weighted hypotheses on the robot's trajectory. Consequently, a different map is estimated for each particle, i.e., for each separate path estimation.

In other words, by treating each particle as the true trajectory, we solve N_p, the mapping with known poses problems, in which all map features are uncorrelated. This enables independent processing of all the feature measurements. In more detail, each map feature is estimated through an EKF, which means that given N_p particles and N_l map features, we have $N_p \times N_l$ different EKFs.

The algorithm pipeline can be subdivided into steps, as summarized in Figure 1. In more detail:

- *Prediction*, where the robot pose is predicted based on the motion commands and a motion model (Section 2.1.1);
- *Data Association*, where the observations are matched with the previously stored map features. This step is not performed if the observations are assumed to be already labeled (*known data association*). In this work, we address the *unknown data association* case (Section 2.2);
- *Proposal Adjustment*, where the robot's pose estimation is adjusted based on the matched observations. This step marks the difference between FastSLAM 1.0 and FastSLAM 2.0 and ensures better robustness in the pose estimate whenever the proprioceptive sensors have higher noise levels than the exteroceptive sensors (Section 2.1.3);
- *Landmark Estimation*, where the pose estimations of the observed landmarks are updated in an EKF-like fashion according to the observations. At this step, weights are assigned to each particle (Section 2.1.4);
- *Importance resampling*, where particle estimations can be maintained or deleted with a probability proportional to their weight. The weight is calculated within the *Landmark Estimation* step. Several strategies are available in the state-of-the-art, allowing for different trade-offs between accuracy and execution time (Section 2.1.5).

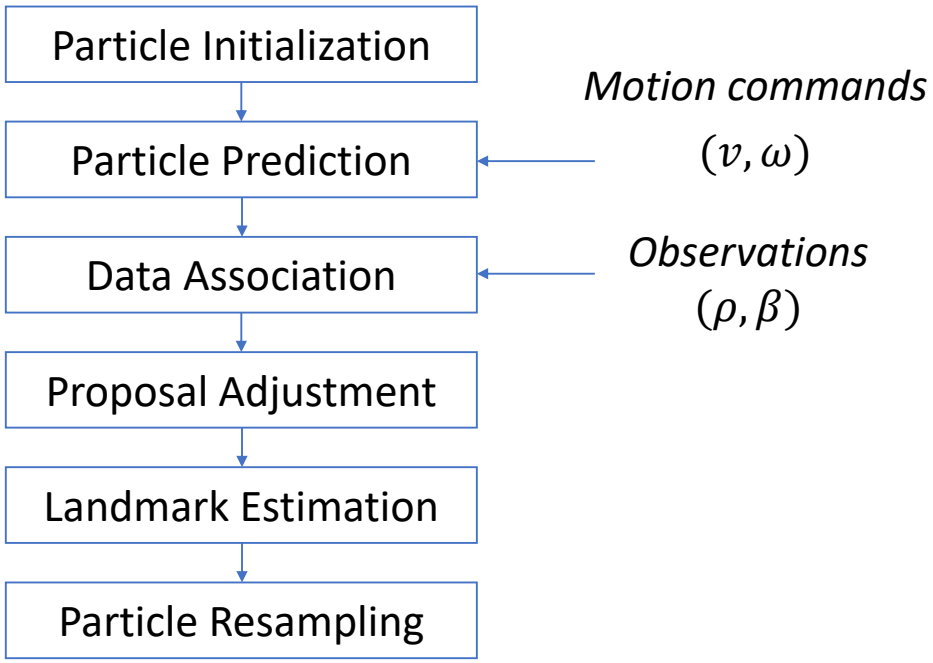


Figure 1. FastSLAM 2.0 pipeline.

2.1.1. Particle Prediction

In this step, an evolution model leverages the control inputs and the old estimation to predict the new robot's pose. For the sake of our dissertation, the control inputs are considered unknown. Therefore, the linear $[m/s]$ and angular velocity $[m/s^2]$ data derived from the simulated inertial data (v, ω) are considered the control input. The prediction step is shown in Algorithm 1.

Algorithm 1 Particle Prediction

for each particle m **do**
  $x_t^m = x_{t-1}^m + v \cos \theta_{t-1}^m \delta t$
  $y_t^m = y_{t-1} + v \sin \theta_{t-1}^m \delta t$
  $\theta_t^m = \theta_t = \theta_{t-1}^m + \omega \delta t$
end for

The complexity of the prediction step depends on the number of particles.

2.1.2. Data Association

After acquiring the exteroceptive data, it is necessary to match the current observations to the features already stored in the map and/or initialize new landmarks. Such a problem can be solved with various approaches depending on the available sensor setup and the navigated scenario. As mentioned in Section 1, in this dissertation, we tackle data association through the JCBB method, a probabilistic approach for jointly associating multiple non-uniquely identifiable observations. Generally, the complexity of the data association step strongly depends on the number of observations and map features. If data association is performed particle-wise, meaning that different particles carry different matches for the same observations, its complexity also depends on the number of particles.

Section 2.2 tackles the data association problem in more detail.

2.1.3. Proposal Adjustment

After the data association step, if one or more observations are matched, the latest robot pose estimation is updated based on the matched observations. The latest pose estimate is modeled as a normal distribution $\mathcal{N}(\mu^m, \Sigma^m)$ and is initialized based on the predicted pose $\mu_0^m = s_t^m$ and a covariance matrix that depends the motion noise Q_t and the Jacobian of the motion model with respect to the control variables H_u.

Subsequently, for each matched observation, the adjustment of the corresponding landmark pose estimate is carried out based on its covariance matrix C^n, the measurement noise R_n, the Jacobians of the observation model with respect to the particle pose H_a and the landmark pose H_p, and the difference between the observation and the expected observation $(z_{n,t} - \hat{z}_{n,t})$ (innovation).

The expected observation $\hat{z}_{n,t}$ is obtained through the observation model in Equation (1).

$$\begin{aligned} dx &= x_l^m - x^m \\ dy &= y_l^m - y^m \\ \hat{\rho} &= \sqrt{dx^2 + dy^2} \\ \hat{\beta} &= \arctan\left(\frac{dy}{dx}\right) \end{aligned} \tag{1}$$

A graphical representation of the observation model is available in Figure 2.

The Jacobian matrix H_a is defined as follows:

$$H_a = \begin{bmatrix} \frac{\partial \rho}{\partial x_l} & \frac{\partial \beta}{\partial x_l} \\ \frac{\partial \rho}{\partial y_l} & \frac{\partial \beta}{\partial y_l} \end{bmatrix} = \begin{bmatrix} \frac{dx}{\sqrt{r}} & \frac{dy}{\sqrt{\hat{\rho}^2}} \\ -\frac{dy}{\sqrt{r}} & \frac{dx}{\sqrt{\hat{\rho}^2}} \end{bmatrix}$$

The matrix H_p is defined as follows:

$$H_p = \begin{bmatrix} \frac{\partial \rho}{\partial x_r} & \frac{\partial \rho}{\partial y_r} & \frac{\partial \rho}{\partial \theta_r} \\ \frac{\partial \beta}{\partial x_r} & -\frac{\partial \beta}{\partial y_r} & \frac{\partial \beta}{\partial \theta_r} \end{bmatrix} = \begin{bmatrix} -\frac{dx}{\sqrt{r}} & -\frac{dy}{\sqrt{\hat{\rho}^2}} & 0 \\ \frac{dy}{\hat{\rho}} & -\frac{dx}{\hat{\rho}} & -1 \end{bmatrix} \tag{2}$$

The procedure for the proposal adjustment is summarized in Algorithm 2.

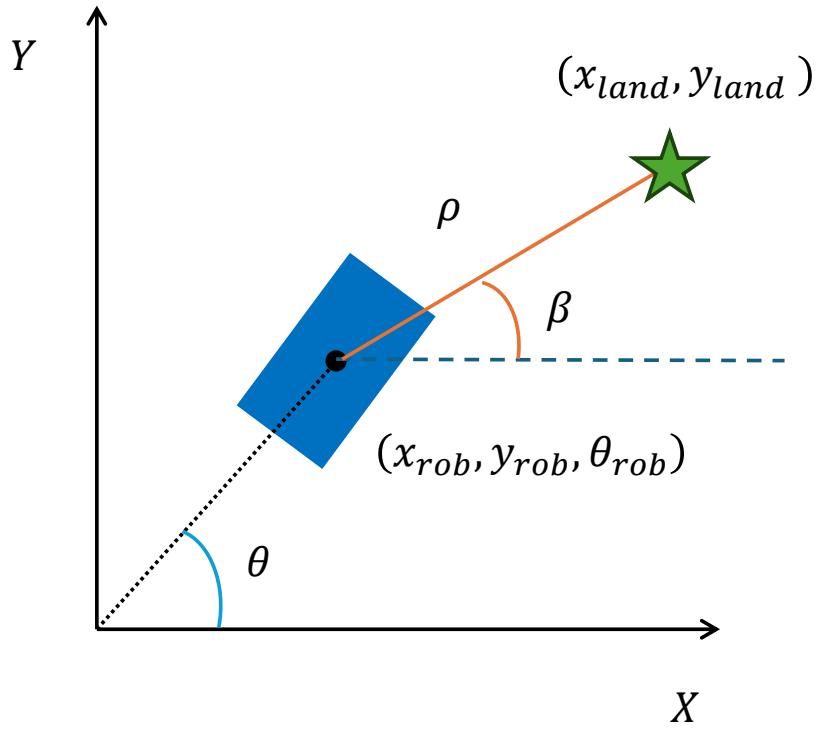


Figure 2. Observation model—graphical representation.

Algorithm 2 Proposal Adjustment

for each particle m **do**
  $\mu_0^m = s_t^m$
  $\Sigma_0^m = H_u Q_t H_u^T$
  for each matched observation n **do**
    $Z_n = H_a C_n H_a^T$
    $\Sigma_n^m = [H_p^T Z_n^{-1} H_p + (\Sigma_{n-1}^m)^{-1}]^{-1}$
    $\mu_n^m = \mu_{n-1}^m + \Sigma_n^m H_p^T Z_n^{-1}(z_{n,t} - \hat{z}_{n,t})$
  end for
  $s_t^n \sim \mathcal{N}(\mu_n^m, \Sigma_n^m)$
end for

The complexity of the proposal adjustment step depends on the number of particles and the number of matched observations.

2.1.4. Landmark Estimation

Given the new proposal distribution, each landmark position in the map is corrected based on the new observations in an EKF-like fashion, meaning that a separate EKF is applied to each observed landmark. New landmarks are initialized based on observations not associated with any preexisting map feature. The initialization happens according to the inverse observation model, which is defined as follows:

$$x_l = x_r + \rho \cos(\theta_r + \beta),$$
$$y_l = y_r + \rho \cos(\theta_r + \beta)$$

The landmark estimation step is summarized in Algorithm 3, where the Jacobian H_a is derived as described in Section 2.1.3.

Algorithm 3 Landmark Estimation

1: **for each** particle m **do**
2:
3: **for each** matched observation n **do**
4: $\hat{z}_{t,n}^m = h(s_t^m X_n^m)$ ▷ Predicted Observation
5: $Y = z_{t,n} - \hat{z}_{t,n}^m$ ▷ Innovation
6: $H_a = \frac{\partial h(s_t^m X_n^m)}{\partial X_n^m}$ ▷ Jacobian
7: $Z = H_a C_n^m H_a^T + R$
8: $K = K_n + K H_a Z^{-1}$ ▷ Kalman Gain
9: $X_n^m = X_n^m + KY$ ▷ Landmark Pose
10: $C_n^m = C_n^m - K H_a C_n^m$ ▷ Landmark Covariance
11: **end for**
12:
13: **for each** unmatched observation n **do**
14: $X_n^m = h^{-1}(z_n^m, s^m)$
15: $\hat{\rho} = \sqrt{(x_n^m - x_r^m)^2 + (y_n^m - y_r^m)^2}$
16: $H_a = \frac{\partial h}{\partial X_n^m}$ ▷ Jacobian
17: $C_n^m = H_a^{-1} Q {H_a^{-1}}^T$
18: **end for**
19:
20: **if** ∃! unmatched observation **then**
21: $w^m = \frac{1}{N_p}$
22: **else**
23: $L_u = H_p Q_t H_p^T + Z$
24: $w^m = \frac{1}{\sqrt{|(2\pi Z)|}} \exp\left(-0.5 Y^T L_u Y\right)$
25: **end if**
26: **end for**

In this step, a weight is assigned to the particles. If at least one unmatched observation exists, the particle weights are reinitialized as in line 21 in Algorithm 3. Otherwise, the weight is assigned according to what is stated on line 24.

The complexity of this step depends on the number of particles and the number of observations.

2.1.5. Particle Resampling

The resampling step corrects the proposal distribution by eliminating or preserving the particles based on their importance weight. This implies that the hypotheses with a higher likelihood have a higher chance of being kept than the least likely. The elimination is carried out probabilistically, and plenty of algorithms are available in the literature, guaranteeing different trade-offs in terms of accuracy and latency. Since our goal is to provide a context-adaptable design for FastSLAM 2.0, we aim to accommodate the broadest possible variety of scenarios. Therefore, based on a dedicated state-of-the-art review, as discussed in Section 2.3, we provide a parallelized implementation of a rather exhaustive pool of resampling methodologies. The user is consequently able to select the most suitable parallelization methodology according to the needs of their use case scenario.

Generally, the complexity of the resampling step depends on the number of particles.

2.2. Data Association

This section is dedicated to a more detailed discussion about the data association step. First, we provide a state-of-the-art overview of the probabilistic data association techniques, focusing on the FastSLAM implementation. Then, Section Joint Compatibility Branch and Bound tackles the Joint Compatibility Branch and Bound (JCBB) method and

its fundamental characteristics to provide the reader with sufficient background on the methodology to understand the following sections of this paper.

The data association task, i.e., the matching between previously observed features and the current observation, is still an open problem in SLAM applications. The problem can be solved deterministically when it is possible to identify the landmarks univocally. For instance, the data association method leveraged in [39] is the FAST corner detector [48], which assigns a unique signature for each landmark by considering its surrounding pixels. However, the data association problem can be solved probabilistically whenever the landmarks are not uniquely identifiable. Since FastSLAM 2.0 is a particle filter-based method, a different data association is carried out for each particle, leading to potentially different association pairings. To the best of our knowledge, no parallel FastSLAM 2.0 implementation on GPU with unknown data association is yet available in the literature. Data association in FastSLAM approaches was addressed for the first time in [49]. Here, maximum likelihood, mutual exclusion, per particle basis, and reuse methods, previously used for EKF-SLAM, are leveraged. Furthermore, the works proposed in [50,51] provide an exhaustive overview of probabilistic data association for particle filter- and EKF-based SLAM approaches. Most of these methodologies address data association referring to Nearest Neighbour-based and Maximum Likelihood-based estimations. The work proposed in [52] proposes Multiple Hypothesis Tracking for data association in particle filters. However, the main limitation of such methods is that they do not guarantee consistent association hypotheses when dealing with multiple observations simultaneously. The Sequential Compatibility Nearest Neighbour algorithm leverages a greedy approach to generate joint hypotheses, with the risk of outputting spurious pairings. The Joint Compatibility Branch and Bound (JCBB) method proposed in [47] addresses the problem of unknown data association with multiple observations by subjecting each joint hypothesis to a compatibility test. More details about JCBB are discussed in Section Joint Compatibility Branch and Bound, together with its adaptation to particle filter-based methods.

Joint Compatibility Branch and Bound

The Joint Compatibility Branch and Bound (JCBB) algorithm [47] was selected to carry out the data association task in our implementation. JCBB can handle multiple observations simultaneously and reduces the number of possible observation–landmark matches by performing a chi-square compatibility test based on the joint quadratic Mahalanobis distance between the observations and the landmarks. After the joint compatibility test is performed, the feasible pairings are evaluated and selected according to the branch-and-bound method. The validity of using a joint compatibility test within the data association step has been demonstrated in [47].

To the best of our knowledge, JCBB has been implemented in the state-of-the-art in combination with the Extended Kalman Filter (EKF) algorithm and not with the Particle Filter (PF) algorithm, which is the basis of FastSLAM 2.0. Since the Mahalanobis distance computation is subject to the assumptions characterizing each estimation method (EKF in the state-of-the-art, PF in our work), it has to be derived accordingly, as shown below. The Mahalanobis distance computed for hypothesis H_i is expressed as follows:

$$D^2_{H_i} = h^T_{H_i} S^{-1}_{H_i} h_{H_i} \tag{3}$$

where h_{H_i} is the difference between the observations and the expected measurement and C_{H_i} is given by the following:

$$S_{H_i} = H_{H_i} P H^T_{H_i} + G_{H_i} R G H_i{}^T \tag{4}$$

where P is the estimation error covariance function for the robot state and the landmark position and H is a matrix of Jacobians discussed later. G is the matrix of Jacobians of the measurement model with respect to the observations. Given our measurement model and

motion model described in Equation (1) and in Algorithm 1, G is an identity matrix. R is the measurement noise covariance matrix. This implies the following:

$$G_{H_i} R G_{H_i}^T = R$$

The first element of Equation (4) has yet to be computed. In the implementations available in the state-of-the-art, which are related to EKF-SLAM, the covariance matrix P looks like Equation (5), where on the main block diagonal we can find the covariance matrix of the robot pose estimation error Cov_r and the landmark pose estimation error covariance matrices Cov_{l_i}, and the generic extra-diagonal elements $A_{i,j}$ represent the correlation between the robot pose and the estimated landmark position (all the $A_{i,1}$, which are non-zero in the EKF), and the landmarks among themselves (zero in any case). However, since in the PF, the landmarks and the robot pose are not correlated conditionally to each particle, P is a block diagonal matrix. Furthermore, since Rao–Blackwallization allows for estimating the landmark poses conditionally to the robot pose, Cov_r is also null conditionally to each particle. Therefore, P is a block diagonal matrix, which is also symmetrical by construction, as in the following:

$$P = \begin{bmatrix} Cov_r & A_{1,2} & A_{1,3} & \cdots & A_{1,N} \\ A_{2,1} & Cov_{l_0} & A_{2,3} & \cdots & A_{2,N} \\ A_{3,1} & A_{3,2} & Cov_{l_1} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & A_{N-1,N} \\ A_{N,1} & A_{N,2} & \cdots & A_{N,N-1} & Cov_{l_n} \end{bmatrix} \tag{5}$$

On the other hand, H generically looks like the matrix below:.

$$H = \begin{bmatrix} H_{p,0} & H_{a,0} & A_{1,2} & A_{1,3} & \cdots & A_{1,N} \\ H_{p,1} & A_{2,1} & H_{a,0} & A_{2,3} & \cdots & A_{2,N} \\ H_{p,2} & A_{3,1} & A_{3,2} & H_{a,0} & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & A_{N-1,N} \\ H_{p,N} & A_{N,1} & A_{N,2} & \cdots & A_{N,N-1} & H_{a,0} \end{bmatrix}$$

where $H_{a,i}$ are the Jacobians of the observation model with respect to the landmark pose and $H_{p,i}$ are the Jacobians of the observation model with respect to the robot pose.

Therefore, the multiplication of HPH^T results in the following:

$$HPH^T = \begin{bmatrix} H_{a_0}^T C_{l_0} H_{a_0} & 0 & 0 & \cdots & 0 \\ 0 & H_{a_1}^T C_{l_1} H_{a_1} & 0 & \cdots & 0 \\ 0 & 0 & H_{a_2}^T C_{l_2} H_{a_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & H_{a_{n-1}}^T C_{l_{n-1}} H_{a_{n-1}} \end{bmatrix}$$

which is a block diagonal matrix, where on the principal diagonal we have 2×2 symmetric matrices. Therefore, the inversion of the matrix is, in turn, a block diagonal matrix whose principal diagonal has the inverse of each covariance matrix. Consequently, the calculation of the joint compatibility as in Equation (3) can be computed in the form of a sum, as in the following:

$$D^2 = \sum_{l=1}^{N_o} h_l^T (H_{a_l}^T C_l H_{a_1})^{-1} h_l$$

2.3. Particle Resampling

This section aims to provide a state-of-the-art overview of the resampling methods and a brief discussion of their advantages and disadvantages. In order to accommodate

different use case scenarios (e.g., accuracy vs. latency trade-offs, characteristics of the hardware in use), a pool of algorithms was selected and included in the parallelized design. The user can adopt the most suitable parallelization modality according to their needs.

A state-of-the-art analysis suggests that many of the available proposals consist of variations of a few methodologies. Consequently, such algorithms are included in our parallelization design to provide a rather exhaustive set of options while keeping generality. The pseudocode illustrating the selected methodologies is available in Appendix A.1 for better manuscript readability. The resampling step occurs after the *Landmark Estimation*, where each particle is assigned a weight. Resampling tends to eliminate particles with lower weights, which have a low likelihood, and replace them with particles with higher weights. After this rearrangement of the particle set, the weights are set to $\frac{1}{N_p}$. Resampling algorithms are intrinsically sequential and might cause a bottleneck in the execution time optimization, which is why their parallelization is widely studied in the literature.

The works proposed in [53–55] provide an exhaustive overview of the available approaches in the current scientific literature. All the algorithms rely on a similar principle: a random number is generated from a uniform distribution between 0 and 1. This number then generates a threshold determining whether a given particle is kept or eliminated based on its weight. The selected algorithms can be subdivided into the following subgroups:

- **Traditional methods**, which, by providing an unbiased proposal, guarantee the best performance in terms of accuracy to the price of efficiency [54]. In particular, the stratified and systematic approaches require iterating among the vector of weights N_p times in the worst-case scenarios. Furthermore, these algorithms usually require computing the inclusive prefix sum of the weights, which can affect the computation time. We selected Multinomial resampling, Stratified resampling, and Systematic resampling among the unbiased methods. Multinomial [56], shown in Algorithm A1, is the most straightforward resampling approach and is based on similar principles as the bootstrap method [57]. In Stratified resampling [58], shown in Algorithm A2, particles are kept or rejected based on N_p randomly generated numbers. In contrast, in systematic resampling [59], shown in Algorithm A3, only a single random number is generated, consequently reducing discrepancy. These algorithms require computing the inclusive prefix sum of the particle weights, leading to an $O(n)$ complexity. These methods are further compared in [60].
- **Alternative methods**, which address the efficiency issues, potentially at the cost of accuracy. These approaches do not require the computation of the inclusive prefix sum and limit the number of iterations among the particle set, requiring tuning a hyperparameter to achieve a satisfactory accuracy vs efficiency trade-off. Rejection resampling, Metropolis resampling, and Coalesced Metropolis (C1 and C2) resampling are selected among these methods. Rejection resampling [61] (Algorithm A4) is an unbiased algorithm and requires the computation of an upper bound for the particle weights. The particle selection is performed within a while loop, making the execution duration of each thread nondeterministic. Furthermore, the upper bound must be very tight to allow the algorithm to perform well. However, the calculation of $w_{max} = max\{w_0 \dots w_n\}$ would introduce a collective operation.
  On the other hand, Metropolis resampling [61] (Algorithm A5) requires the generation of numerous random numbers, but the tuning of a hyperparameter B binds the thread execution time. However, if B is set to an excessively low value, the sample is biased, harming convergence. According to [53], the improved methods Metropolis C1 (Algorithm A6) and Metropolis C2 (Algorithm A7), introduced in [62], achieve better time performance in shared memory architectures and for a very high number of particles by spanning among a chunk of the particle set instead of the whole set under the assumption that accessing adjacent or close elements in an array is beneficial for time efficiency. In this case, a further hyperparameter has to be tuned to determine the number of intervals.

In [53], the resampling performance was assessed over 100 executions with over 2^{16} particles.

3. Hardware and Software Setup

This section describes the hardware and software setup leveraged for the deployment of the proposed design and the validation of the evaluation methodology described in Section 4. The proposed parallelization of FastSLAM 2.0 is based on the CUDA programming paradigm and uses PyCUDA [63] for Python interfacing. The performance was tested on a high-end laptop equipped with an Intel Core i7-10750H CPU [63] and an NVIDIA GeForce GTX 1650 Ti [63].

3.1. Hardware Setup

The hardware used to deploy and validate our design was a high-end laptop hosting an Intel Core i7-10750H CPU and an NVIDIA GeForce GTX 1650 Ti. The specifications of the CPU architecture are summarized in Table 1.

Table 1. Specifications of Intel Core i7-10750H CPU.

Processor	Intel Core i7-10750H
Base Clock Speed	2.60 GHz
Max Turbo Frequency	Up to 5.00 GHz
Number of Cores	6
Number of Threads	12
Lithography	14 nm
Cache	12 MB Intel Smart Cache
Memory Type Support	DDR4-2933
Max Memory Size	64 GB
Integrated Graphics	Intel UHD Graphics
TDP	45 W
Socket	FCLGA1200

The NVIDIA GeForce GTX 1650 Ti is based on the Turing architecture and features 1024 CUDA cores optimized for parallel processing tasks. Designed primarily for gaming laptops and compact desktops, this graphics processing unit (GPU) integrates several features that enhance its suitability for algorithm acceleration in computational research. The main specifications of the NVIDIA GeForce GTX 1650 Ti are gathered in Table 2.

Table 2. Specifications of NVIDIA GeForce GTX 1650 Ti.

Architecture	Turing Architecture (Turing TU117 GPU)
Single Multiprocessors	16
CUDA Cores	1024
Base Clock	1350 MHz
Boost Clock	1485 MHz
Memory	GDDR6 Memory with Varying Capacities (typically 4GB)
Memory Interface	128-bit
Memory Bandwidth	Varies, typically around 192 GB/s
TDP	Around 55–80 W

Table 2. *Cont.*

Max Blocks per Grid	$2^{31} - 1$
Max Threads per Block	1024
Max Shared Memory per Block	48 KB (49,152 bytes)

3.2. Cuda Programming Paradigm

High-performance computing has gained rising interest in the past few years, and modern Graphic Processing Units (GPUs) are at the center of this trend due to the possibility of heavily parallelizing computation in large-scale applications.

In this context, NVIDIA developed CUDA [64], a programming model and software environment that allows programmers to write scalable codes, also called kernels, in a C-like programming language [65]. In more detail, a kernel is a small program that runs on the GPU rather than the CPU. It is a fundamental concept in parallel computing and is used to execute a portion of a program on the GPU. In CUDA, a kernel is executed by multiple threads on the GPU. Threads are the basic execution units that execute the same kernel code but with different data. A group of threads cooperatively executing the same kernel is called a *block*. Blocks are usually mapped to the single multiprocessors (SMs) and are divided into subsets of 32 threads called *warps*.Warps are launched randomly within the same block. The execution context within each warp is maintained on-chip. While the number of threads per warp is constant among all GPU architectures (i.e., 32), the number of threads per block and the number of blocks in the grid depend on each specific device.

The threads belonging to the same block can cooperate by sharing data through shared memory. The shared memory is a small, fast memory space that is shared among threads within a block. It is divided into 32 banks, each of which can be accessed simultaneously by a warp. In contrast, global device memory is a larger, slower memory space shared among all GPU threads. While global device memory provides a larger storage capacity, it is slower than shared memory due to its higher latency.

When being launched, each block is assigned to a single multiprocessor (SM), a processing unit on the GPU that executes a block of threads. Each multiprocessor has its own memory, registers, and execution units. The threads in a block are executed concurrently on the multiprocessor, allowing for efficient parallel processing of data. However, to ensure correct execution, it is necessary to synchronize the threads. This is achieved through the use of barriers, which force threads to wait until all threads in the same block have reached a certain point before proceeding to the next instruction. A set of blocks executed on the GPU is called a *grid*. The blocks in a grid run independently, which means that block synchronization is not contemplated within the CUDA programming paradigm. This is achieved through the use of multiple multiprocessors on the GPU, each of which can execute a block of threads concurrently.

When a CUDA kernel is launched, the following steps occur: the host (CPU) launches a kernel, specifying the number of blocks and threads per block; the GPU schedules the blocks and assigns them to available multiprocessors; each block is executed on a single multiprocessor, where the threads in the block are executed concurrently; the threads in a block cooperate with each other, sharing data and synchronizing their execution; and the blocks in a grid are executed independently.

3.3. PyCUDA

PyCUDA [63] is an open-source toolkit that supports Runtime Code Generation for GPUs. It allows the programmer to adopt a scripting-based approach when programming GPUs. Every feature of CUDA runtime is accessible through PyCUDA, while memory allocation and resource management are handled automatically as in a high-level programming language. However, it is possible to deallocate memory in applications with tight memory usage manually. PyCUDA allows for the creation of GPU binaries by providing CUDA source code. This architecture enables the adoption of an edit–run–repeat working style,

which is quite comfortable for exploratory prototyping and full-scale code, too, thanks to the use of the GPU. Therefore, the Python code running on the CPU executes control and communication tasks that require high abstraction. In contrast, the C-like code running on the GPU solves low-level and throughput-oriented problems.

In our implementation, PyCUDA enables the launching of the CUDA kernels and the memory transfer from host to device.

3.4. Memory Access Management

Our parallelization of FastSLAM 2.0 is designed to run on a heterogeneous architecture, where the CPU and the GPU, which can be defined as *host* and *device*, respectively, are equipped with two separate memories and execute commands asynchronously. The parallelized FastSLAM 2.0 is interfaced with a simulator, described in Section 4.1, that runs on the CPU and feeds the algorithm with synthetic sensor data. The CPU can also allocate memory on the GPU to store variables and launch kernels. All the FastSLAM 2.0 pipeline operations leverage either the GPU global memory or the on-chip memory (shared memory and local thread memory). The final results are eventually copied to the host device (CPU). A schema of the memory management is provided in Figure 3.

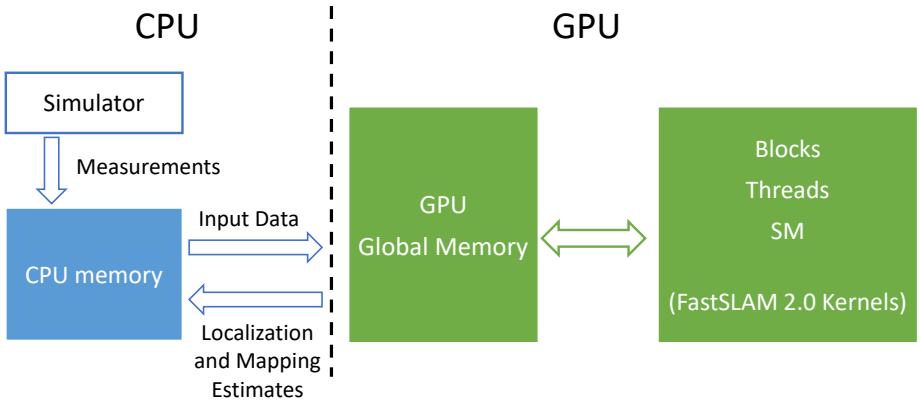


Figure 3. Hardware–software architecture schema.

The proposed implementation of FastSLAM 2.0 optimizes memory access to accelerate processing time. This is achieved by limiting access to the global device memory whenever possible in favor of shared memory when cooperative data manipulation is needed, as well as local thread memory.

Shared memory in a GPU is fast, on-chip memory accessible by all threads within the same thread block. It enables efficient data sharing among threads and significantly improves the performance of parallel algorithms by allowing data reuse and reducing redundant calculations.

While shared memory is much faster than global memory, offering latency and bandwidth comparable to registers, it is limited in size and can vary depending on each GPU architecture. The GPU architecture where we validate our approach is equipped with 48 kilobytes of shared memory per block. All threads within the same block can read from and write on shared memory. Shared memory is divided into banks. The bank size and the number of banks depend on each specific GPU architecture. If multiple threads belonging to the same warp access the same bank simultaneously, bank conflicts may arise, degrading performance. The memory bank conflict is commonly avoided through common CUDA programming good practices such as padding and accurate offset selection to ensure that all the threads in the same block contemporarily access different banks. Furthermore, conflicts can arise when thread divergence occurs. Therefore, it is utterly important to correctly execute thread synchronization to ensure that the read and write operations on the shared memory are coherently performed.

4. Evaluation Methodology

This section discusses the evaluation methodology we propose to adapt and optimize the deployment of FastSLAM 2.0 onto the hardware architecture in use. Firstly, we analyze the algorithm's dependencies. Then, we divide the algorithm pipeline into functional blocks with the same dependencies. Subsequently, each parameter is thresholded to limit the processing time. The parallelized design of each functional block contemplates one or more parallelization modalities, depending on which parameters affect its complexity (i.e., number of particles N_p, number of existing landmarks N_l, number of observations N_o, and number of matched observations N_c), as further described in Section 5.

Dependency analysis and parameter thresholding are widely employed in the parallelization process of different types of SLAM algorithms. In [66,67], such a methodology is leveraged to study the performance of graph-based SLAM on embedded systems; in [68], it was used to evaluate an efficient implementation of EKF-SLAM, and in [39], it was used for FastSLAM 2.0.

Our design takes a comprehensive approach by exploring multiple parallelization methodologies to achieve the best performance across different contexts. We leverage a dependency analysis to define multiple possible grid configurations, which are later deployed on the hardware architecture of interest. The evaluation is carried out by comparing each parallelization modality to the sequential implementation according to the performance indexes described in Section 4.4. The performance analysis under different test cases (i.e., contexts) is carried out in the form of a sensitivity analysis, which means that the variation of the index of interest is observed as an effect of the variation of a single parameter, while the others are kept constant.

In the state-of-the-art, the parallelization modality is not always disclosed, making it rather impractical to perform our validation methodology on such proposals with the available information. However, whenever such information is available, the parallelization usually happens particle-wise (i.e., the number of concurrent operations is equal to N_p). Due to the adaptability of our design, we can test multiple modalities, including, among others, particle-wise parallelization. Furthermore, we wish to remark that, unlike the state-of-the-art proposals, it is not within our scope to prove the optimality of a single implementation under the widest variety of scenarios but to provide a flexible design that can be easily adapted to reach optimality under various conditions and with the support of various hardware architectures.

4.1. Simulation Environment

The simulation environment we leverage to benchmark our design models the robot as a rigid body—i.e., it is characterized by its Cartesian coordinates and its orientation (x_r, y_r, θ_r) and it moves in a bi-dimensional space characterized by landmarks. The landmarks are modeled as points in space and are defined by their Cartesian coordinates and a label $(x_l, y_l, label)$. The robot is considered equipped with a generic range sensor and an inertial sensor. The sensors are characterized by a measurement noise modeled as zero-mean Gaussian white noise.

As for the inertial sensors, linear and rotational speed (v, ω) are generated with the following:

$$v_{meas} = v_{true} + \mathcal{N}(0, \sigma_v^2)$$
$$\omega_{meas} = \omega_{true} + \mathcal{N}(0, \sigma_\omega^2)$$

and the range and bearing signals (ρ, β) coming from the range sensors are computed as

$$\hat{\rho} = \sqrt{(x_l - x_r)^2 + (y_l - y_r)^2} + \mathcal{N}(0, \sigma_\rho^2)$$
$$\hat{\beta} = \arctan\left(\frac{y_l - y_r}{x_l - x_r}\right) - \theta_r + \mathcal{N}(0, \sigma_\omega^2)$$

The simulation environment runs on CPU and is schematized in Figure 4.

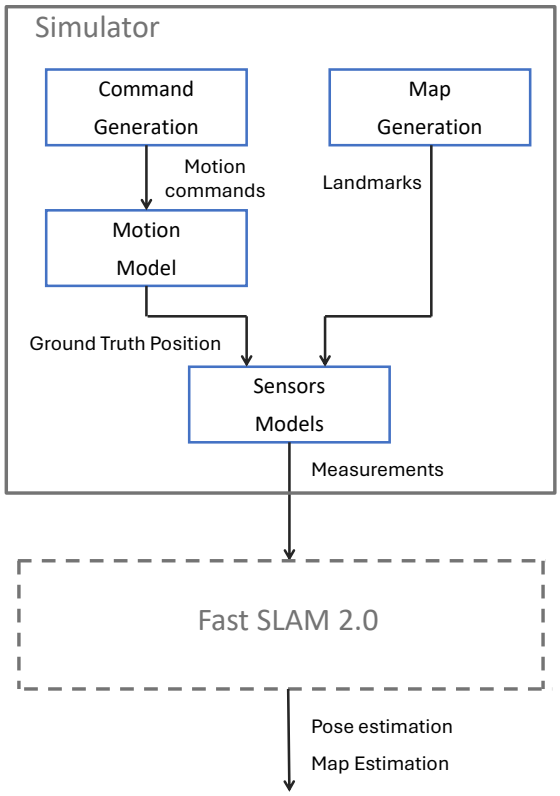


Figure 4. Simulation environment schema.

4.2. Functional Block Partitioning

As mentioned in Section 4, we analyze the algorithm based on the instruction and operation order, as in [39]. We then divide the pipeline into functional blocks as summarized in Figure 5. The first functional block, dedicated to the *Particle Initialization* step, initializes the particle poses with random numbers and the landmarks as null values. The output of the *Particle Initialization* block constitutes the input of the *Particle Prediction* functional block, together with the inertial sensor signals from the simulator. This functional block is supposed to run only at the first iteration of the algorithm. The predicted poses and the unlabeled observations retrieved from the simulator are the input of the *Data Association* functional block. The associated observations outputted by the *Data Association* block and the pose prediction are fed to the *Proposal Adjustment* block, which outputs the updated robot pose estimate. The output of this functional block and the associated observations are then fed to the *Landmarks Update and Weights Computation* block. The corrected landmark estimates, the corrected particle pose, and the newly assigned weights feed the *Resampling* functional block, which is responsible for copying the selected particles to the new particle set and re-initializing the particle weights. The output of the *Resampling* blocks, together with the new signals coming from the inertial sensors, will constitute the input for the *Particle Prediction* and *Data Association* functional blocks for the next iteration.

All the functional blocks are composed of a single kernel except for the *Data Association* functional block, which is subdivided into multiple kernels. Further implementation details are discussed in Section 5.

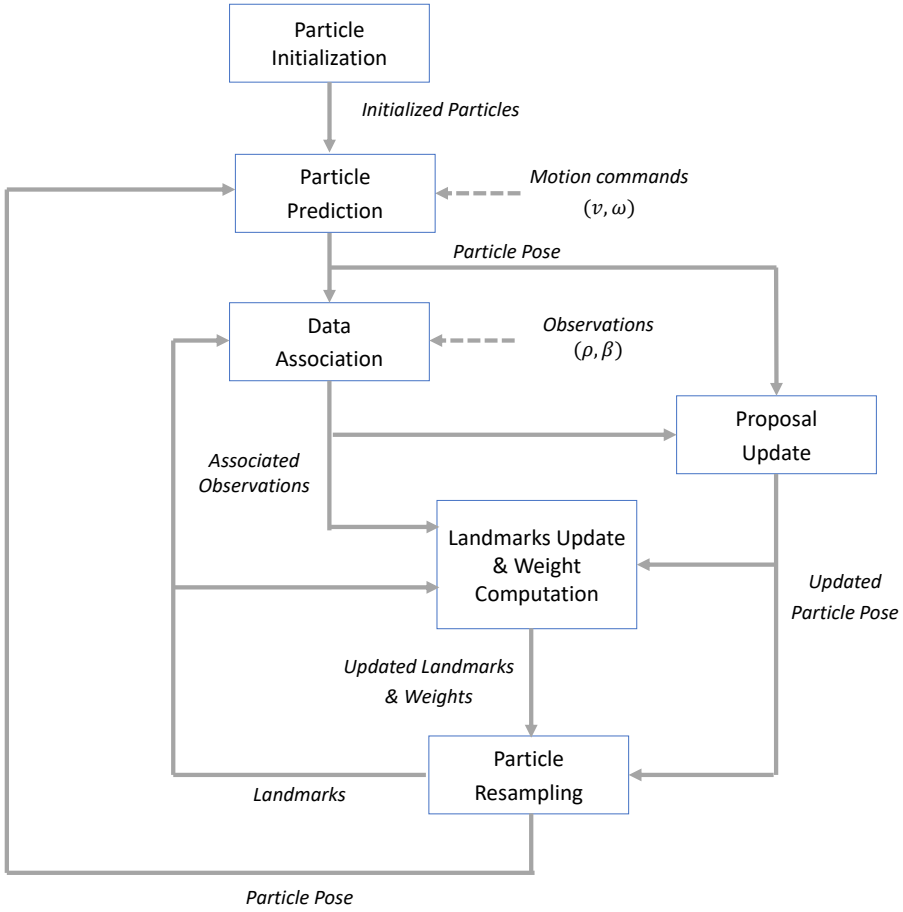


Figure 5. Functional blocks partitioning schema.

4.3. Algorithm Dependencies and Definition of Test Cases

After the algorithm has been analyzed and divided into functional blocks, we can identify their dependencies. By *dependencies*, we mean the variables that affect the computational complexity of each block. The functional blocks discussed in Section 4.2 coincide with the steps described in Section 2.1. Their dependencies are summarized in Table 3.

Table 3. Algorithm dependencies.

Step	N_p	N_l	N_o
Particle Initialization	✓	✗	✗
Particle Prediction	✓	✗	✗
Data Association	✓	✓	✓
Proposal Adjustment	✓	✗	✓
Landmark Estimation	✓	✗	✓
Resampling	✓	✗	✗

Based on the dependency analysis, we can finally determine the parallelization possibilities for each functional block. In particular, the parallelization modalities are reported below:

- *Particle Initialization*: The parallelization is performed based on the particles.
- *Particle Prediction*: The parallelization is performed based on the particles.
- *Data Association*: The parallelization is performed based on the particles, on the previously observed landmarks, and jointly on the particles and the landmarks. Although the complexity of the data association step also depends on the number of observa-

tions, the necessity of performing feasible joint hypotheses (i.e., each observation is associated with at most one landmark) forces the developer to tackle them sequentially.
- *Proposal Adjustment*: The parallelization is performed based on the particles, on the matched observations, and jointly on the particles and the matched observations.
- *Landmark Estimation and Weight Update*: The parallelization is performed based on the particles, on the total observations, and jointly on the particles and the observations.
- *Importance Resampling*: The parallelization is performed based on the particles.

Such parallelization modalities are then mapped onto the grid combination options available in the GPU programming paradigm:

- Sequential—one block, one thread ($1B1T$)
- One dimension, thread-based—one block, N threads ($1BNT$)
- One dimension, block-bases—M blocks, one thread ($MB1T$)
- Two dimensions—M blocks, N threads ($MBNT$).

The sensitivity analysis is carried out by varying one of the dependencies and keeping all the others constant. We perform such evaluation for all the dependencies in each step, except for the number of observations in the *Data Association* functional block. This is because JCBB builds the association hypotheses incrementally, which implies a sequential execution on the observations. Consequently, it is impossible to assess the benefit of parallelization based on this parameter.

Each dependency is assigned a threshold, as shown in Table 4. These thresholds are generally defined with the sole scope of binding the execution time. However, further limitations may arise depending on the memory availability of each specific device and the memory requirements of each kernel. For the sake of the completeness of our sensitivity analysis, we apply the aforementioned thresholds when possible. However, in an actual use case scenario, it is necessary to find an optimal combination between the memory availability of the hardware device, the desired precision of the estimation (proportional to the number of particles), and the map's scale.

Table 4. Dependency thresholds.

Dependency	Threshold
N_p	4096
N_l	4096
N_o	1024

4.4. Performance Assessment

The sensitivity analysis described in the previous sections for the performance assessment is based on two parameters: the *Elapsed Time* and *Time Gain*. The Elapsed Time is computed for each functional block B, for each context C (i.e., combination of parameters based on which the sensitivity analysis is performed), and for each grid combination G as the average of $N = 500$ repeated executions, as follows:

$$\text{Elapsed Time}(B, C, G) = \sum_{n=1}^{N} \frac{t(B, C, G, n)}{N}$$

and the Time Gain is defined as the ratio between the sequential grid combination (1B1T) and the most advantageous grid combination, as

$$\text{Time Gain}(B, C, G^{best}) = \frac{ElapsedTime(B, C, G^{1B1T})}{ElapsedTime(B, C, G^{best})}$$

The *Elapsed Time* quantity is expressed in milliseconds ([ms]), while the Time Gain is scalar.

5. Deployment

This section discusses the design details of each parallelized functional block. Figure 6 shows how the memory transfer and the interaction between CPU and GPU are managed across the entire pipeline. The dashed rectangles correspond to data, while the colored rectangles contain instructions (CUDA kernels and their launchers).

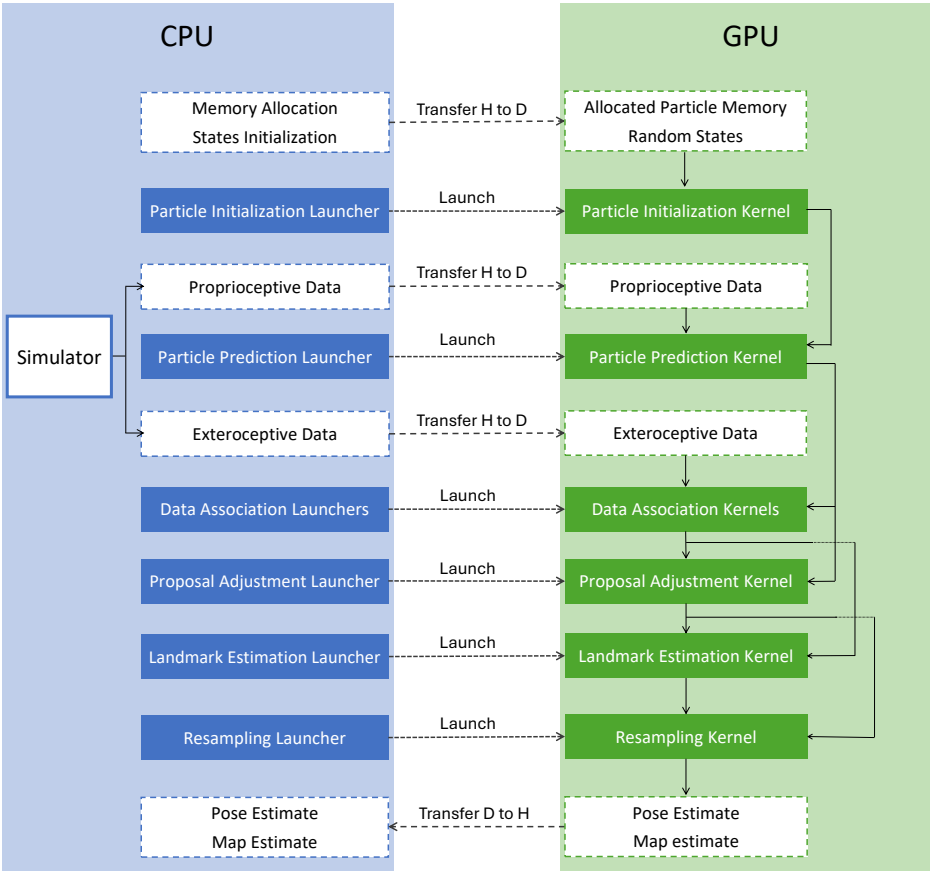


Figure 6. Detailed heterogeneous architecture pipeline.

In more detail, the CPU allocates memory on the GPU and initializes a vector of random states leveraged in the random number generation operations. Then, each functional block is implemented in the form of one or two kernels that start their execution on the GPU once a dedicated launcher is executed. The interaction within the kernels is akin to what is described in Figure 5. Once all the operations are terminated, the pose estimates and the map estimates are copied back to the host memory to be fed as input to the *Particle Prediction Launcher* at the next iteration.

The pseudocode relevant to the parallelization of the functional blocks in the following sections is available in Appendices A.2–A.6. This section is aimed at discussing the design peculiarities of our approach; for the methodological and mathematical foundations of FastSLAM 2.0, please refer to Section 2.

5.1. Particle Initialization

The *Particle Initialization* is parallelized based on the number of particles. Therefore, a monodimensional parallelization is performed. Algorithm A8 iterates over each particle with a stride corresponding to $blockDim \cdot gridDim$. However, regarding the mapping of this monodimensional parallelization onto grid composition possibilities, it is possible to launch the kernel with the following:

- Sequential configuration ($1B1T$): one block and one thread.

- Thread-based configuration ($1BNT$): one block and $min(N_p, 1024)$ threads.
- Block-based configuration ($MB1T$): N_p blocks and one thread.
- Two-dimension configuration ($MBNT$): Upper Bound ($\frac{N_p}{1024}$) blocks and 1024 threads.

In this case, the memory limitation is determined by the amount of global memory available in the GPU.

5.2. Particle Prediction

The *Particle Prediction* is parallelized based on the number of particles. Therefore, we perform a monodimensional parallelization. Algorithm A9 iterates over each particle with a stride corresponding to $blockDim \cdot gridDim$. However, regarding the mapping of this monodimensional parallelization onto grid composition possibilities, it is possible to launch the kernel with the following:

- Sequential configuration ($1B1T$): one block and one thread
- Thread-based configuration ($1BNT$): one block and $min(N_p, 1024)$ threads.
- Block-based configuration ($MB1T$): N_p blocks and one thread
- Two-dimension configuration ($MBNT$): Upper Bound ($\frac{N_p}{1024}$) blocks and 1024 threads.

As in the *Particle Prediction step*, the memory limitation is determined by the amount of global memory available in the GPU.

5.3. Data Association

The *Data Association* step is subdivided into three kernels, as summarized in Figure 7. The pseudocode for the data association step is available in Appendix A.4 for better manuscript readability. Though a single-kernel implementation is possible, the high amount of needed shared memory would heavily limit the map's scale and the precision of the estimation (proportional to the number of particles). Therefore, with the goal of achieving a trade-off between the increase in performance given by the usage of the faster on-chip memory and the limitation given by its small size, we privilege the shared memory usage for those variables that are read and written more than once. In particular, the variables related to the individual compatibility assessment would need to be stored in the shared memory together with the expected observations $\hat{z}_n^m$. However, the high dimensionality of the first variables (highlighted in Algorithm A10) and the reduced number of times they need to be accessed made a multiple-kernel implementation preferable, limiting the shared memory usage for the expected observations. Therefore, the JCBB operations are divided as follows. A first kernel, available in Algorithm A10, is in charge of computing the quantity $H_a \cdot C_n^m \cdot H_a^T$ derived in Section Joint Compatibility Branch and Bound for each possible observation–landmark pair and performing an individual compatibility test. The resulting matrix is, therefore, stored in the global memory and constitutes the exploration tree where branch-and-bound is carried out. Each element in the cost matrix $MD_{i,j}$ represents the Mahalanobis distance between observation i and map feature j. If the individual compatibility test is not passed, $MD_{i,j}$ is set to infinity.

Therefore, all the feasible hypotheses are explored while all the non-feasible matches are excluded. An intermediate kernel called *Problem Preparation*, shown in Algorithm A9, arranges some auxiliary parameters to optimize tree exploration by excluding unmatchable observations (i.e., where none of the $MD_{\bar{i},j}$ for all j passed the individual compatibility test) and initializes new labels for the newly observed landmarks. In this case, the rows of the cost matrix corresponding to non-matchable observations are not explored.

Eventually, the *Branch and Bound* kernel explores the feasible combinations and assesses the most promising hypotheses, as shown in Algorithm A12. Every time a new observation is considered, a joint compatibility test is performed for each feasible partial hypothesis (i.e., the most promising partial hypothesis outputted by the previous iteration in addition to the Mahalanobis distance between the new observation and each not-yet-matched map feature). If the compatibility test is passed, the lower bound for the partial hypothesis is computed. Otherwise, the cost is set to infinity. As a last step, the minimum cost for

the partial hypothesis is computed through parallel reduction and the most promising combination is selected.

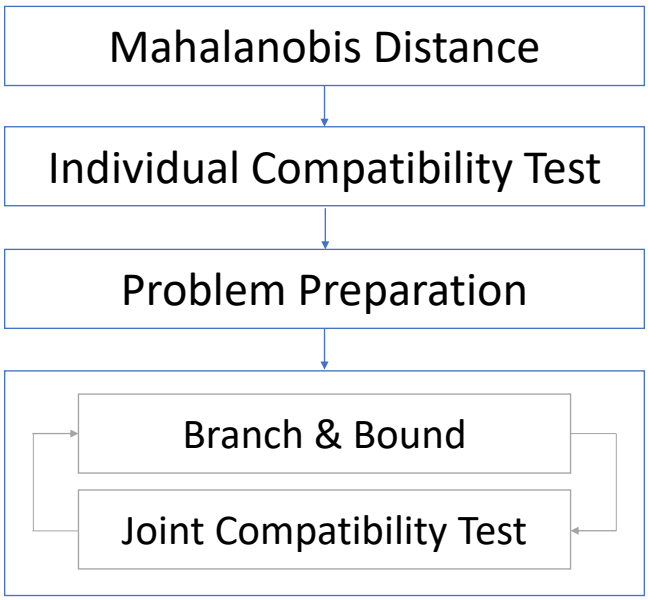


Figure 7. Data association pipeline.

The complexity of the data association step depends on the number of particles, the number of previously observed landmarks, and the number of observations, but, as mentioned in Section 4.3, the parallelization is performed on the particles and the landmarks. The choice is motivated by the consideration that JCBB builds the association hypothesis incrementally, implying that the observations must be iterated sequentially. Furthermore, in an average scenario, the number of observations is much lower than the number of landmarks already present in the map. This implies that the most significant benefit is given by performing particle-wise and landmark-wise parallelization. Therefore, monodimensional and bi-dimensional parallelizations are performed. An external for loop iterates over the particles and an inner loop iterates over the landmarks. Consequently, the different parallelization modalities are achieved according to the following grid configurations:

- Sequential configuration ($1B1T$): one block and one thread.
- Thread-based configuration ($1BNT$): one block and $min(N_o, 1024)$ threads. Parallel on the landmarks and sequential on the particles and the observations.
- Block-based configuration ($MB1T$): N_p blocks and one thread. Parallel on the particles and sequential on the landmarks and observations.
- Two-dimension configuration ($MBNT$): N_p blocks and $min(N_o, 1024)$ threads. Parallel on the particles and the landmarks and sequential on the observations.

The three kernels store their intermediate outputs in the global memory. Due to the high dimensionality of these outputs, the main limitation in terms of memory availability is the size of the device's global memory.

5.4. Proposal Adjustment

The *Proposal Adjustment* functional block depends on the number of particles and matched observations. Therefore, monodimensional and bi-dimensional parallelizations are possible. This is achieved by assigning particles to blocks and observations to threads. The choice is motivated by the necessity of collaborative data management within the same particle. Therefore, an external loop iterates over the particles, and an internal loop over the observations. Consequently, the different parallelization modalities are achieved according to the following grid configurations:

- Sequential configuration ($1B1T$): one block and one thread.
- Thread-based configuration ($1BNT$): one block and $min(N_o, 1024)$ threads—parallel on the observations and sequential on the particles.
- Block-based configuration ($MB1T$): N_p blocks and one thread—parallel on the particles and sequential on the observations.
- Two-dimension configuration ($MBNT$): N_p blocks and $min(N_o, 1024)$ threads—parallel on the particles and the observations.

Algorithm 4 summarizes the proposal adjustment step. Each line corresponds to a block that is further expanded in Algorithms A13–A17 in Appendix A.5.

Algorithm 4 Proposal Adjustment

1: **for** $m = blockId$ to N_p, step = $gridDim$ **do**
2: Initialization Σ_0, μ_0
3: Preparation for adjustment
4: Σ_n computation
5: μ correction computation
6: Correction application
7: **end for**

Each block carries out a separate operation on the shared memory, which is why threads are synchronized at the end of each block. In particular, the shared memory variables, highlighted with comments in the relative algorithms, are related to the cumulative computation of the correction applied to the robot's pose estimate based on the matched observations. In Algorithm A16, the innovation is computed for a second time to accommodate the cases in which $N_o > 1024$ because each block would tackle multiple observations sequentially, and the innovation values would be overwritten in the local memory threads.

The cumulative prefix sums in Algorithms A16 and A17 are carried out through stride-like parallel reduction to avoid bank conflicts as in [69].

This functional block significantly uses the shared memory. Therefore, the main limitation in memory availability resides in the shared memory per block.

5.5. Landmark Estimation

The *Landmark Estimation* functional block depends on the number of particles and matched observations. Therefore, monodimensional and bi-dimensional parallelizations are feasible. In more detail, the particles are assigned to blocks, and threads are assigned to the observations. The choice is motivated by the fact that the reinitialization of the weights depends on all the matched observations within each particle. Therefore, the external for loop iterates over the particles and an inner loop iterates over the observations. Consequently, multiple parallelization modalities are achievable with the following grid configurations:

- Sequential configuration ($1B1T$): one block and one thread.
- Thread-based configuration ($1BNT$): one block and $min(N_o, 1024)$ threads—parallel on the observations and sequential on the particles.
- Block-based configuration ($MB1T$): N_p blocks and one thread—parallel on the particles, sequential on the observations.
- Two-dimension configuration ($MBNT$): N_p blocks and $min(N_o, 1024)$ threads—parallel on the particles and the observations.

The landmark estimation is carried out on a single kernel. However, as in the previous case, the algorithm is split into different sections for better readability. A summary of the landmark estimation step is available in Algorithm 5. Please refer to Appendix A.6 for more details.

The main limitation in memory availability is the GPU's global memory, as the kernel uses a limited amount of shared memory.

Algorithm 5 Landmark Estimation

```
1: for m = blockId to N_p, step = gridDim do
2:     for n = threadId to N_o, step = blockDim do
3:         Retrieve observation z_n
4:         if new landmark then
5:             Landmark Initialization
6:         else
7:             Landmark Update
8:         end if
9:     end for
10:    Weight Update
11: end for
```

5.6. Particle Resampling

The *Particle Resampling* is parallelized based on the number of particles. Therefore, a monodimensional parallelization is performed. Algorithm 6 shows a general template for the resampling step. Each tested resampling variation contains a different selection strategy, as reported in Line 4. The particle weights are preliminarily copied onto the shared memory to optimize memory access. Our implementation iterates over each particle with a stride corresponding to $blockDim \cdot gridDim$. Therefore, it is possible to launch the kernel with the following:

- Sequential configuration ($1B1T$): one block and one thread.
- Thread-based configuration ($1BNT$): one block and $min(N_p, 1024)$ threads.
- Block-based configuration ($MB1T$): N_p blocks and one thread.
- Two-dimension configuration ($MBNT$): Upper Bound ($\frac{N_p}{1024}$) blocks and 1024 threads.

Algorithm 6 Importance Resampling

```
1: Retrieve W                                          ▷ vector of weights, Shared Memory
2: W = ∑_{j=0}^{N} w_j, j = 0, ..., N_p                ▷ traditional methods only, Shared Memory
3: for m = gridId to N_p, step = blockDim · gridDim do
4:     Choose selectedIdx
5:     Retrieve (x_r^selectedIdx, y_r^selectedIdx, θ_r^selectedIdx)      ▷ to Thread
6:     Transfer (x_r^selectedIdx, y_r^selectedIdx, θ_r^selectedIdx)      ▷ to Global Memory
7:     Retrieve map features for selectedIdx-th particle                 ▷ to Thread
8:     Transfer map features for selectedIdx-th particle                 ▷ to Global Memory
9: end for
```

The cumulative prefix sum is computed with parallel reduction as in Section 5.4 for the traditional methods. As for rejection resampling, w_{max} is computed as $max\{w_0, \ dotsw_N\}$, meaning that one collective operation is introduced. This is because we do not assume that the particle weights are normalized, which makes the estimation of an upper bound hard to formulate. On the other hand, a work-efficient parallel reduction based on [69,70] is implemented for the maximum computation. Unlike the inclusive prefix sum, which requires iterating over the particles twice, such an implementation requires iterating only once, increasing the efficiency of our implementation of rejection resampling in comparison with traditional methods.

In the case of resampling, the memory limitation is determined by the amount of shared memory available in the GPU.

6. Results

In this section, the experimental results are presented and discussed. In particular, we show the results of the sensitivity analyses described in Section 4 and draw some conclusions on the performance of the flexible design under different use case scenarios.

Sections 6.1–6.6 tackle the results of the individual functional blocks, while Section 6.7 discusses our results on a more general level.

6.1. Initialization

The comparison of the parallelization modalities discussed in Section 5.1 is shown in Figure 8. The plot shows that for a low number of particles (i.e., <1024), block-based parallelization ($MB1T$) is to be preferred. In contrast, the best performance for higher numbers of particles is given by limiting the number of blocks and maximizing their occupancy ($MBNT$). On the other hand, the $MB1T$ configuration is the most inconvenient parallelization modality for a higher number of particles due to the overhead involved when launching a high amount of blocks. The $MBNT$ configuration shows constant processing time for all the tested use cases where the number of particles exceeds 1024.

The most efficient parallel implementation guarantees an acceleration of a factor of 60 for 1024 particles and around 250 for 4096 particles.

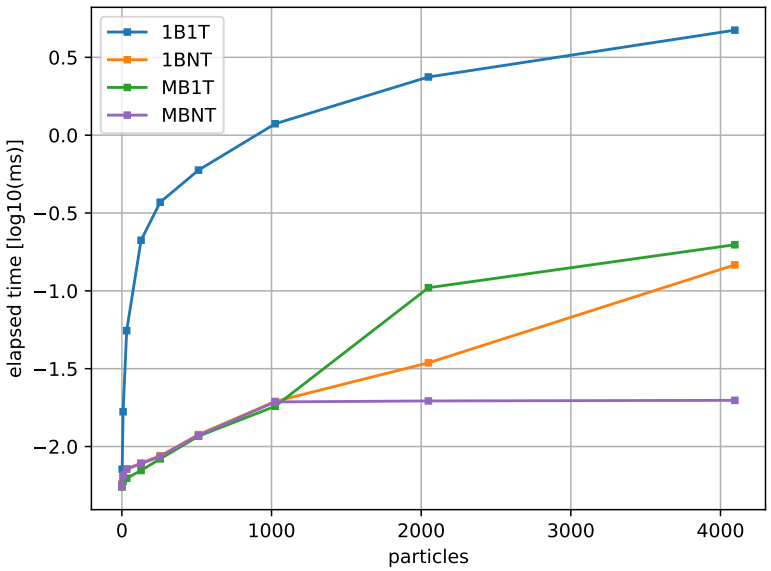


Figure 8. Particle Initialization—elapsed time.

6.2. Prediction

The comparison of the experimental results for the parallelization modalities discussed in Section 5.2 is displayed in Figure 9. For a low number of particles (i.e., <1024), the block-based parallelization (MB1T) shows the best performance. In contrast, limiting the number of blocks by maximizing their occupancy ($MBNT$) minimizes the processing time for higher numbers of particles. The $1BNT$ configuration is the most inconvenient parallelization strategy. The $MBNT$ configuration shows constant processing time for all the tested use cases for numbers of particles above 1024.

The most efficient parallel implementation guarantees an acceleration of a factor of 70 for 1024 particles and around 270 for 4096 particles.

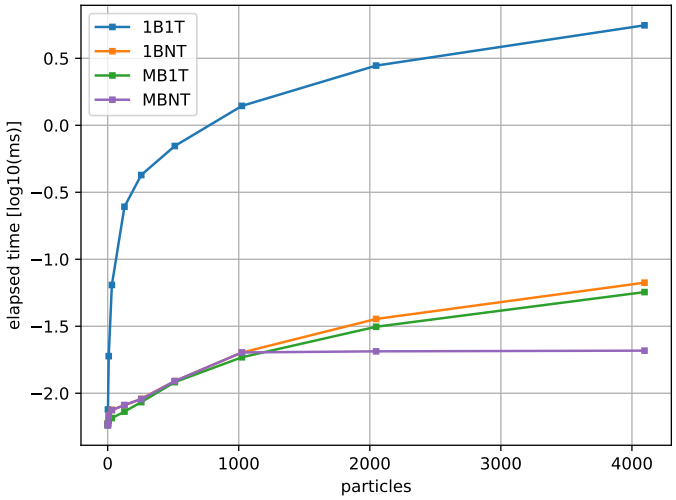


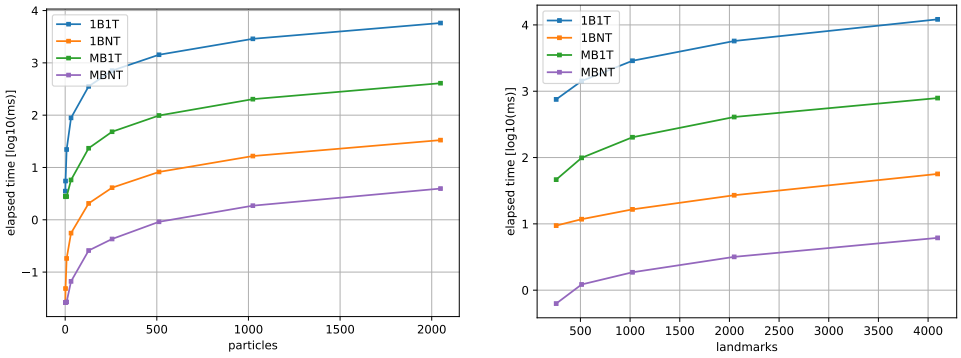
Figure 9. Particle Prediction—elapsed time.

6.3. Data Association

The data association functional block is subdivided into three kernels as discussed in Section 5.3. Therefore, the experimental results are shown for the three kernels separately in Sections 6.3.1–6.3.3. The sensitivity analysis is performed based on the number of particles and landmarks on the map. Since our implementation deals with the observations sequentially, this parameter is kept constant at 16.

6.3.1. Mahalanobis Distance

The experimental results of the sensitivity analyses for the Mahalanobis Distance computation are shown in Figure 10. Figure 10a shows the results of the particle-based sensitivity analysis ($N_l = 1024$), while Figure 10b the results for the landmark-based analysis ($N_p = 1024$). The fully parallelized implementation ($MBNT$) performs best under all test cases since the Mahalanobis Distance computation is highly computationally intensive on the landmarks, making block-based parallelization the most inefficient among the parallelized implementations. The most efficient implementations reduce by a factor of around 1500 for the highest numbers of particles ($N_l = 1024$), a factor of almost 2000 for the highest values of N_l, with $N_p = 1024$.



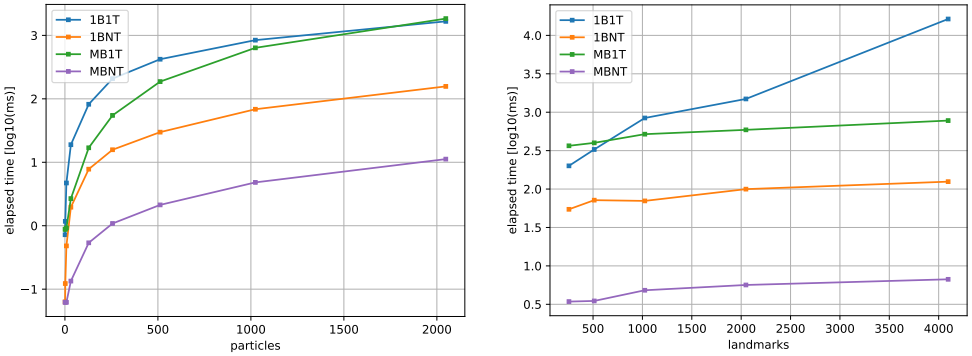
(**a**) By particles (**b**) By landmarks

Figure 10. Mahalanobis Distance—elapsed time.

6.3.2. Problem Preparation

The experimental results of the sensitivity analyses for the Problem Preparation are shown in Figure 11. Figure 11a shows the results of the particle-based sensitivity analysis ($N_l = 1024$), while Figure 11b shows the results for the landmark-based analysis

($N_p = 1024$). The fully parallelized implementation ($MBNT$) demonstrates the best performance among all test cases. The most efficient implementations reduce the processing time by a factor of around 150 for the highest numbers of particles ($N_l = 1024$), a factor of almost 260 for the highest values of N_l, with $N_p = 1024$.



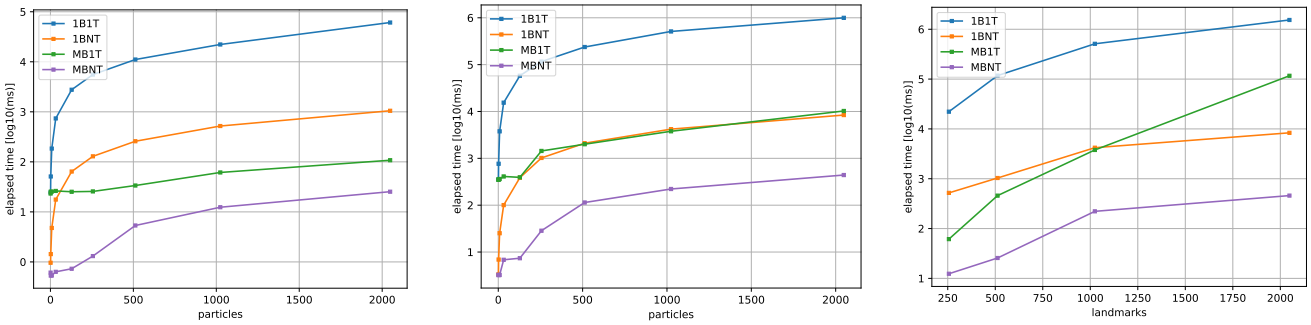
(**a**) By particles (**b**) By landmarks

Figure 11. Problem Preparation—elapsed time.

6.3.3. Branch and Bound

The experimental results of the sensitivity analyses for the Branch and Bound are shown in Figure 12. Figure 12a,b show the results of the particle-based sensitivity analysis with $N_l = 256$ and $N_l = 1024$ respectively, while Figure 11b shows the results for the landmark-based analysis ($N_p = 1024$). The fully parallelized implementation ($MBNT$) demonstrates the best performance among all test cases. It is worth pointing out that for a low number of landmarks, the block-based ($MB1T$) parallelization performs better than the thread-based parallelization ($1BNT$), which, in turn, is more efficient for a higher number of landmarks.

The most efficient implementation reduces the processing time by a factor of around 2500 for the highest numbers of particles ($N_l = 1024$), a factor of almost 220 for the highest values of N_l, with $N_p = 1024$.



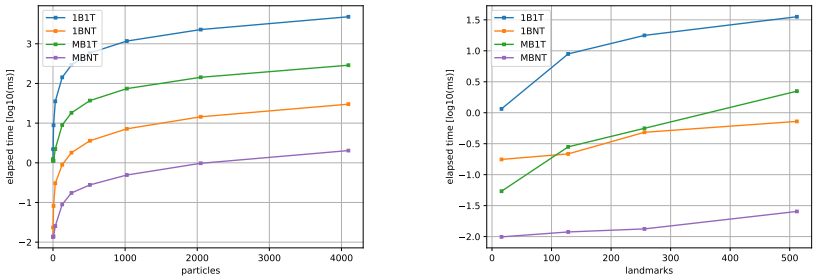
(**a**) By particle—256 landmarks (**b**) By particle—1024 landmarks (**c**) By landmark

Figure 12. Branch and Bound—elapsed time.

6.4. Proposal Adjustment

The experimental results of the sensitivity analyses for the Proposal Adjustment are shown in Figure 13. Figure 13a shows the results of the particle-based sensitivity analysis ($N_l = 512$), while Figure 13b shows the results for the observed landmark-based analysis ($N_p = 1024$). The fully parallelized implementation ($MBNT$) demonstrates the best performance among all test cases. It is worth pointing out that for a low number of landmarks, the block-based (MB1T) parallelization performs better than the thread-based parallelization ($1BNT$), which, in turn, is more efficient for a higher number of landmarks.

The most efficient implementation reduces the processing time by around 2300 for the highest numbers of particles ($N_l = 512$).



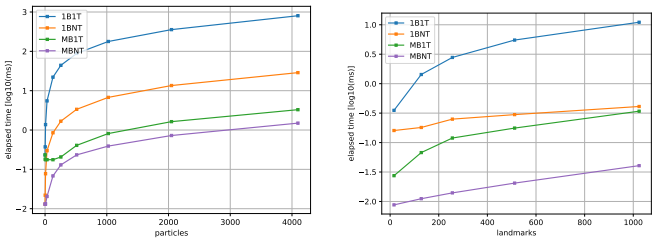
(**a**) By particles (**b**) By landmarks

Figure 13. Proposal Adjustment—elapsed time.

6.5. Landmark Estimation

The experimental results of the sensitivity analyses for the *Landmark Estimation* are shown in Figure 14. Figure 14a shows the results of the particle-based sensitivity analysis ($N_l = 512$), while Figure 14b shows the results for the observed landmark-based analysis ($N_p = 1024$). The fully parallelized implementation ($MBNT$) demonstrates the best performance among all test cases. For a low number of landmarks, thread-based ($MB1T$) parallelization is the least efficient among the parallel implementations.

The most efficient implementation reduces the processing time by a factor of around 250 for the highest numbers of particles ($N_l = 512$), a factor of almost 270 for the highest values of N_l, with $N_p = 1024$.



(**a**) By particles (**b**) By landmarks

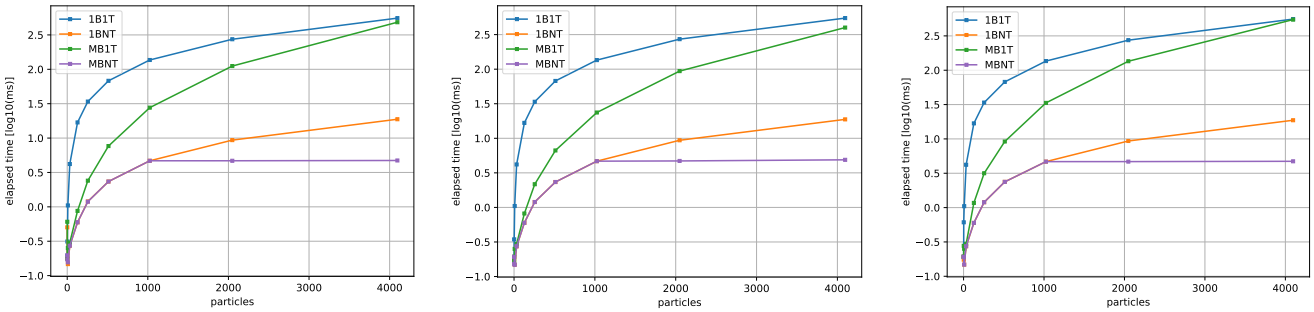
Figure 14. Landmark Estimation—elapsed time.

6.6. Particle Resampling

The results for the sensitivity analysis for the traditional resampling methods are shown in Figure 15 and in Figure 16 for the alternative resampling methods. The coalesced Metropolis algorithm is executed by setting the number of iterations to 10 and a segment size of 32 if $N_p > 32$ and of N_p in the other cases. As for the traditional resampling methods, the $MBNT$ configuration proves to be the most efficient in all the examined cases, while the block-based parallelization ($MB1T$) shows the least efficiency among the parallel implementations, reaching the same elapsed time as the sequential implementation in the case of multinomial resampling (Figure 15a) and systematic resampling (Figure 15c). This is due to the need to compute the cumulative prefix sum performed sequentially in all blocks, which heavily increases the processing time.

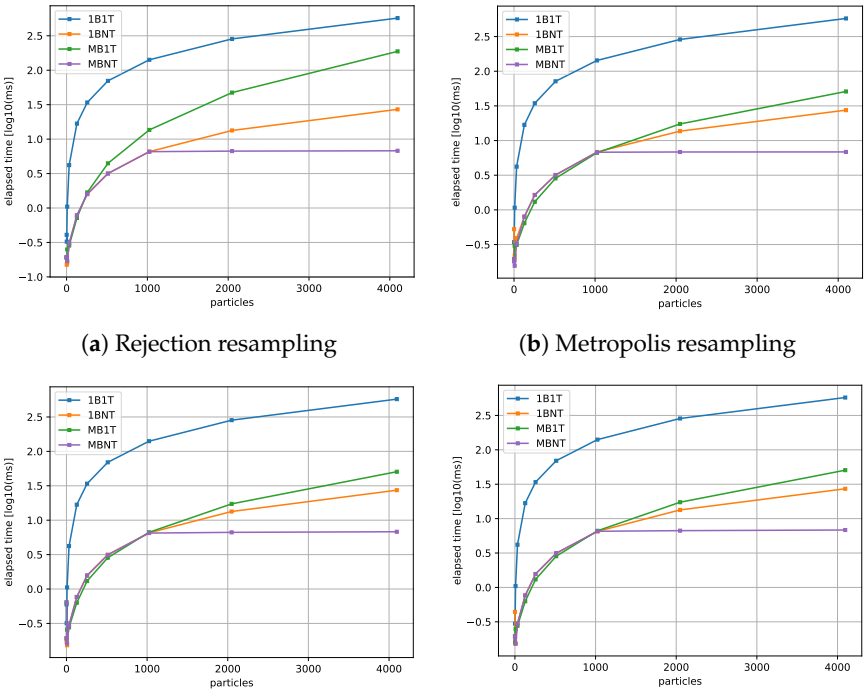
As for the alternative methods, which do not require the computation of the inclusive prefix sum, we can observe a slightly different behavior. In the case of the Metropolis methods (Metropolis resampling in Figure 16b, Metropolis C1 in Figure 16c, and Metropolis C2 in Figure 16d), the $MB1T$ implementation is the most efficient for lower numbers of particles, while the $MBNT$ configuration is to be preferred for a higher number of particles. In these three cases, the $MB1T$ configuration is still the most inefficient for a

higher number of particles due to the high overhead implied by launching a significant amount of blocks. The rejection resampling algorithm (Figure 16a) shows a similar behavior as the traditional methods, where the $MBNT$ configuration is preferable under all test cases, and the $MB1T$ configuration is more sensitive to the increase of the number of particles in terms of increased processing time. This is due to the necessity of computing the maximum weight w_{max}, computed via a parallel reduction in every block.



(**a**) Multinomial resampling (**b**) Stratified resampling (**c**) Systematic resampling

Figure 15. Resampling traditional methods—elapsed time.



(**a**) Rejection resampling (**b**) Metropolis resampling

(**c**) Metropolis C1 resampling (**d**) Metropolis C2 resampling

Figure 16. Resampling alternative methods—elapsed time.

For higher numbers of particles under our test cases, the traditional methods are accelerated by 120, while the traditional methods are accelerated by a factor of 80 for the respective most efficient implementations.

6.7. Discussion

In this section, we address some overall considerations on the performance of our flexible design under different contexts (i.e., sets of parameters) regarding the GPU architecture leveraged for the validation of our method.

Firstly, we notice that for the functional blocks requiring monodimensional parallelization and that do not include a high number of operations (e.g., *Particle Initialization*

(Section 6.1) and *Particle Prediction* (Section 6.2)), the $MB1T$ parallelization is ideal for lower values of N_p, while, for higher numbers of particles, it is preferable to limit the number of launched blocks while prioritizing a small grid maximizing the capacity of the single blocks.

Secondly, in the case of those kernels that are richer in operations and that require a bi-dimensional parallelization (depending on Np and N_l), such as the *Mahalanobis Distance* kernel (Section 6.3.1), the fully parallel modality is by far the most preferable under all the cases considered in the landmark-based sensitivity analysis (Figure 10b) and in most of the cases considered in the particle-based sensitivity analysis (Figure 10a). For small values on N_p, the single-block implementation is to be preferred. This is observable because, for the particle-wise sensitivity analysis, the number of landmarks set to $N_l = 1024$ (textiti.e., full block occupancy) and, with the number of particles being quite low, the latency cost of launching multiple blocks at their full capacity is higher than the time gain guaranteed by the parallelization. Similar considerations can be highlighted for the *Problem Preparation*, *Branch and Bound*, *Proposal Adjustment*, and *Landmark Estimation* kernels (Sections 6.3.2–6.4). The *Branch and Bound* kernel had a slightly different behavior in the particle-based sensitivity analysis performed with a lower number of landmarks (Figure 12a). In this case, for lower values of N_p, the fully parallel modality is still the preferable option. This is because blocks are not launched at the maximum of their capacity. The time gain provided by higher levels of parallelization has more impact than the latency given by the launch of the blocks.

As for the resampling algorithms, discussed in Section 6.6, the $MBNT$ configuration is to be preferred under all test cases for the traditional methods (Figure 15). In fact, although they support only particle-wise parallelization like the *Particle Prediction* and *Particle Initialization* kernels, they still require the computation of a collective operation, which would be carried out sequentially in the $MB1T$ configuration, decreasing its performance even for smaller values of N_p. On the other hand, this is not the case for the Metropolis methods (Figures 16), which show instead a similar behavior to the other two functional blocks. On the other hand, the rejection resampling algorithm (Figure 16a) shows a similar behavior as the traditional methods, where the $MBNT$ configuration is preferable under all test cases, and the $MB1T$ configuration is more sensitive to the increase in N_p in terms of processing time. This is due to the necessity of computing the maximum weight w_{max}, computed via a parallel reduction in every block. It is worth remarking that, for higher numbers of particles, the $MBNT$ configuration settles on a lower value for the traditional methods in comparison with the alternative methods. These results seem to contradict the conclusions drawn in the state-of-the-art [53]. In fact, the alternative methods should benefit more from the parallel programming due to the removal of collective operations and, in the cases of Metropolis C1 and Metropolis C2, by the access of more compact portions of the shared memory. However, the test cases mentioned in Section 2.3 concern a very high number of particles ($>2^{16}$) while, in the case of our target GPU architecture, we perform our evaluation on much lower values of N_p. On the other hand, the elimination of the collective operations and the subdivision of the shared memory into intervals introduce the necessity for additional expensive operations such as random number generations. Based on our experimental results, we can conclude that for the values of N_p compatible with the use cases contemplated here and in the case of our specific target architecture, we cannot observe the benefits of optimizations such as coalesced memory access and elimination of collective operations. Therefore, given these specific circumstances, we would not need to compromise on accuracy to reduce the computation time, and we may choose one of the three traditional methods as a resampling algorithm for hypothetical real navigation.

7. Conclusions

In this paper, we proposed a context-adaptable design of FastSLAM 2.0 based on the CUDA parallel computing platform. The proposed design can accommodate different parallelization modalities without the need for a complete re-implementation. Multiple

resampling methodologies have been included in the design to meet the needs of the widest possible variety of use case scenarios. In addition, we provided a methodology to assess the optimal parallelization modality depending on the GPU architecture in use and the specifications of the contemplated navigation use case. Such specifications define the set of parameters determining the algorithm's computational complexity and, as a direct consequence, its execution latency. The algorithm was divided into functional blocks; the parameters affecting their computational complexity were identified and leveraged in the parallelization design, which uses shared memory to enhance collaborative data management. The methodology was validated on a high-end GPU, where all the parallelization modalities were tested under different parameter sets through a sensitivity analysis. As a further contribution, our design includes the parallelization of the data association step, implemented via the Joint Compatibility Branch and Bound (JCBB) methodology.

This flexible implementation of FastSLAM 2.0 and its corresponding selection methodology are designed to facilitate optimization in the deployment process of SLAM onto GPGPU, while enabling real-time robot localization and environmental mapping. Therefore, future work will explore the interfacing of the proposed design with a real environmental sensing setup within a real navigation scenario.

In addition, further adaptations of the proposed design will be contemplated in the scope of future work. Firstly, our design is meant to be compatible with a wide variety of hardware architectures where the CPU (host) and GPU (device) do not share the same memory space and, within the device, the global device memory and the shared memory are also located in separate locations. Examples of GPU memory architectures meeting these characteristics include, but are not limited to, Video Random Access Memory (VRAM), Graphics Double Data Rate (GDDR), and High-bandwidth Memory (HBM). Future developments of our contribution may involve extending the proposed method to architectures such as the Unified Memory Architecture (UMA), which is typical of devices such as the NVIDIA Jetson Nano, which are commonly used for robotics applications. In such devices, the shared memory is a part of the global device memory instead of a separate memory space. Such characteristics would require a redesign of the memory management criteria.

Furthermore, another possible extension of the proposed design is to contemplate 3D pose estimation. As described in Section 4.1, our design models the robot as a rigid body moving across a planar space and the landmarks as points. While such simplification is feasible under many use cases (e.g., autonomous cars, ground robots) and is indeed to be preferred for avoiding adding unnecessary computational complexity, some scenarios might need pose estimation in the tri-dimensional space (e.g., aerial drones, underwater robots). Though necessary changes would not affect the evaluation methodology and the overall parallelization design, they would require an expansion of the state variables, increasing the memory occupancy. As a direct consequence, the hardware architecture in use might enable the support of a smaller map (i.e., less landmarks) and/or a smaller number of particles.

Author Contributions: Conceptualization, J.G. and M.P.C.; methodology, J.G. and M.P.C.; software, J.G.; validation, J.G.; formal analysis, J.G.; investigation, J.G.; resources, J.G. and D.P.M.S.; data curation, J.G.; writing—original draft preparation, J.G.; writing—review and editing, J.G. and M.P.C.; visualization, J.G. and M.P.C.; supervision, M.P.C. and D.P.M.S.; project administration, D.P.M.S.; funding acquisition, D.P.M.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the German Federal Ministry of Education and Research BMBF under grant number 16ME0097 (ZuSE KI-mobil).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: Jessica Giovagnola was employed by the company Infineon Technologies AG. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
EKF	Extended Kalman Filter
FAST	Features from Accelerated Segment Test
GDDR	Graphics Double Data Rate
GNSS	Global Nation Satellite System
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GPS	Global Positioning System
HBM	High-Bandwidth Memory
IF	Information Filter
IMU	Inertial Measurement Unit
JCBB	Joint Compatibility Branch and Bound
LiDAR	Light Detection And Range
PF	Particle Filter
RBPF	Rao–Blackwellized Particle Filter
RTK	Real-Time Kinematics
SLAM	Simultaneous Localization and Mapping
UKF	Unscented Kalman Filter
UMA	Unified Memory Architecture
VRAM	Video Random Access Memory

Appendix A

Appendix A.1. Resampling Algorithms

Algorithm A1 Multinomial Ancestors

function MULTINOMIAL_ANCESTORS($w \in [0, \infty)^N$)
  $W \leftarrow \text{InclusivePrefixSum}(w)$
  for $i \leftarrow 1$ to N **do**
    $u^i \sim U[0, W_N)$ ▷ Sample uniformly in range $[0, W_N)$
    $a^i \leftarrow \text{LowerBound}(W, u^i)$ ▷ Find lower bound index
  end for
  return a
end function

Algorithm A2 Stratified Resampling

function STRATIFIED_RESAMPLING($w \in [0, \infty)^N$)
  $N \leftarrow \text{count}(w)$
  $W \leftarrow \text{InclusivePrefixSum}(w)$
  for $n \leftarrow 1$ to N **do**
    $u_n \sim U[0, 1)$ ▷ Sample uniformly in range $[0, 1)$
  end for
  $u_i \leftarrow \frac{i-1+u_i}{N}$ for $i = 1, \ldots, N$
  $k \leftarrow 1$
  for $i \leftarrow 1$ to N **do**
    while $W(k) < u(i)$ **do**
      $k \leftarrow k + 1$
    end while
    $a(i) \leftarrow k$
  end for
  return a
end function

Algorithm A3 Systematic Resampling

function SYSTEMATIC_RESAMPLING($w \in [0, \infty)^N$)
  $N \leftarrow \text{count}(w)$
  $W \leftarrow \text{InclusivePrefixSum}(w)$
  $u_0 \sim U[0, 1)$ ▷ Sample uniformly in range $[0, 1)$
  $u \leftarrow \frac{u_0}{N}$
  $k \leftarrow 1$
  for $i \leftarrow 1$ to N **do**
    while $W(k) < u(i)$ **do**
      $k \leftarrow k + 1$
    end while
    $a(i) \leftarrow k$
  end for
  return a
end function

Algorithm A4 Rejection Resampling

function METROPOLISC1_RESAMPLING($w \in [0, \infty)^N$)
  $N \leftarrow \text{count}(w)$
  Initialize a as an array of size N
  for $i \leftarrow 1$ to N **do**
    $p \leftarrow i$
    $u \sim U[0, 1)$
    while $u < \frac{w(p)}{w_{max}}$ **do**
      $p \leftarrow U\{1 \dots N\}$
      $u \sim U[0, 1)$
    end while
    $a(i) \leftarrow p$
  end for
  return a
end function

Algorithm A5 Metropolis Resampling

function METROPOLISC1_RESAMPLING($w \in [0, \infty)^N$)
  $N \leftarrow \text{count}(w)$
  $B \leftarrow$ number of iterations
  Initialize a as an array of size N
  for $i \leftarrow 1$ to N **do**
    $p \leftarrow i$
    for $j \leftarrow 1$ to B **do**
      $u \sim U[0, 1)$ ▷ Sample uniformly in range $[0, 1)$
      $q \sim U\{1, \dots N\}$ ▷ Sample within segment
      if $u \leq \frac{w(q)}{w(p)}$ **then**
        $p \leftarrow q$
      end if
    end for
    $a(i) \leftarrow p$
  end for
  return a
end function

Algorithm A6 MetropolisC1 Resampling

function METROPOLISC1_RESAMPLING($w \in [0,\infty)^N$)
  $N \leftarrow \text{count}(w)$
  $B \leftarrow$ number of iterations
  $SC \leftarrow$ number of segments ▷ Number of segments
  $DC \leftarrow$ segment size ▷ Size of each segment
  Initialize a as an array of size N
  for $i \leftarrow 1$ to N **do**
    $p \leftarrow i$
    $s \sim U\{1,\ldots,SC\}$ ▷ Sample segment index
    for $j \leftarrow 1$ to B **do**
      $u \sim U[0,1)$ ▷ Sample uniformly in range $[0,1)$
      $q \sim U\{(s-1)\times DC+1, s\times DC\}$ ▷ Sample within segment
      if $u \leq \frac{w(q)}{w(p)}$ **then**
        $p \leftarrow q$
      end if
    end for
    $a(i) \leftarrow p$
  end for
  return a
end function

Algorithm A7 MetropolisC2 Resampling

function METROPOLISC2_RESAMPLING($w \in [0,\infty)^N$)
  $N \leftarrow \text{count}(w)$
  $B \leftarrow$ number of iterations
  $SC \leftarrow$ number of segments ▷ Number of segments
  $DC \leftarrow$ segment size ▷ Size of each segment
  Initialize a as an array of size N
  for $i \leftarrow 1$ to N **do**
    $p \leftarrow i$
    for $j \leftarrow 1$ to B **do**
      $u \sim U[0,1)$ ▷ Sample uniformly in range $[0,1)$
      $s \sim U\{1,\ldots,SC\}$ ▷ Sample segment index
      $q \sim U\{(s-1)\times DC+1, s\times DC\}$ ▷ Sample within segment
      if $u \leq \frac{w(q)}{w(p)}$ **then**
        $p \leftarrow q$
      end if
    end for
    $a(i) \leftarrow p$
  end for
  return a
end function

Appendix A.2. Particle Initialization

Algorithm A8 Particle Initialization

for $m = threadId$ to N_p, step = $blockDim \cdot gridDim$ **do**
  $x_t^m = randU(x_{min}, x_{max})$
  $y_t^m = randU(y_{min}, y_{max})$
  $\theta_t^m = randU(\theta_{min}, \theta_{max})$
end for

Appendix A.3. Particle Prediction

Algorithm A9 Particle Prediction

for $m = threadId$ to N_p, step = $blockDim \cdot gridDim$ **do**
  $x_t^m = x_{t-1}^m + \delta_s \cdot \cos(\theta_{t-1}^m + \frac{\delta_\theta}{2})$
  $y_t^m = y_{t-1}^m + \delta_s \cdot \sin(\theta_{t-1}^m + \frac{\delta_\theta}{2})$
  $\theta_t^m = \theta_{t-1}^m + \delta_\theta$
end for

Appendix A.4. Data Association

Algorithm A10 Mahalanobis Distance and Individual Compatibility Test

for $m = blockId$ to N_p, step = $gridDim$ **do**
  for $n = threadId$ to N_l, step = $blockDim$ **do**
    $dx = x_n^m - x_r^m$ ▷ position of the landmark w.r.t. particle pose
    $dy = y_n^m - y_r^m$
    $\hat{\rho}^2 = dx^2 + dy^2$
    $\hat{\rho} = \sqrt{\hat{\rho}^2}$ ▷ Expected Range
    $angle = \arctan(\frac{dy}{dx})$
    $\hat{\beta} = angle - \theta_r^m$ ▷ Expected Bearing
    $\hat{z}_n^m = \begin{pmatrix} \hat{\rho} \\ \hat{\beta} \end{pmatrix}$ ▷ Shared Memory
    $H_a = \begin{pmatrix} \frac{dx}{\rho} & \frac{dy}{\rho} \\ \frac{-dy}{\rho^2} & \frac{dx}{\rho^2} \end{pmatrix}$ ▷ Jacobian
    $HPH_n^m = H_a \cdot C_n^m \cdot H_a^T$ ▷ Shared Memory
  end for
  threadsynch()
  for $n = threadId$ to N_l, step = $blockDim$ **do**
    for $k = 0$ to N_o **do** ▷ Sequential
      $dz = z_k - \hat{z}_n^m$ ▷ Innovation
      $S = HPH_n^m + R$
      $MD_{n,k}^m = S^{-1} dz^2$ ▷ Global Memory
      if $MD_{n,k}^m < \chi^2_{2,.95}$ **then**
        $IC_{n,k}^m = 1$ ▷ Global Memory
      else
        $IC_{n,k}^m = 0$ ▷ Global Memory
      end if
    end for
  end for
  threadsynch();
end for

Algorithm A11 Problem Preparation

for $m = blockId$ to N_p, step = $gridDim$ **do**
  for $k = 0$ to N_o **do** ▷ Sequential
    matchable$_k$=CumulativeSum(IC_k) ▷ From Global Memory
    threadSynch();
  end for
  $new = N_o$ ▷ New Label
  $next = 0$ ▷ Auxiliary Variable
  $n_c = 0$ ▷ Global Memory
  for $k = 0$ to N_o **do** ▷ Sequential
    if not matchable$_k$ **then**
      $h_k = new$ ▷ Global Memory
      $new+ = 1$
    else
      $n_c+ = 1$
      $aux_{next} = k$
      $next+ = 1$
    end if
    threadSynch();
  end for
end for

Algorithm A12 Branch and Bound and Joint Compatibility Test

for $m = blockId$ to N_p, step = $gridDim$ **do**
  $cost = 0$
  for $k = 0$ to N_o **do** ▷ Sequential
    $obsId = aux_k$ ▷ Retrieve corresponding matched observation
    for $n = threadId$ to N_l, step = $blockDim$ **do**
      if not assigned$_n$ **then**
        $LB_n^{obsId} = cost + MD^m_{n,obsId}$
      else
        $LB_n^{obsId} = \inf$
      end if
    end for
    threadSynch();
    for $n = threadId$ to N_l, step = $blockDim$ **do**
      if $LB_n^{obsId} < \chi^2_{2k,90}$ **then** ▷ Joint Compatibility Test
        compute PC ▷ Partial Cost
      else
        $PC = \inf$
      end if
      $LB_n^{obsId} + = \inf$
    end for
    threadSynch();
    selectedIdx= $\arg\min(LB)$
    $cost+ = MH^m_{k,selectedIdx}$
    $assigned_{selectedIdx} = 0$
  end for
end for

Appendix A.5. Proposal Adjustment

Algorithm A13 Initialization Σ_0, μ_0

$$\mu_0^m = \begin{pmatrix} x_t^m \\ y_t^m \\ \theta_t^m \end{pmatrix}$$

Algorithm A13 *Cont.*

$H_u = \begin{pmatrix} -\sin(v) & 0 \\ \cos(v) & 0 \\ 0 & 1 \end{pmatrix}$ ▷ Compute Jacobian
$\Sigma_0 = H_u \cdot Q_t \cdot H_u^T$
threadSynch()

Algorithm A14 Preparation for adjustment

for $n = threadId$ to N_c, step = $blockDim$ **do**
  $dx = x_n - x_r$
  $dy = y_n - y_r$
  $angle = \arctan(\frac{dy}{dx} - \theta)$
  $r = (x_l - x_r)^2 + (y_l - y_r)^2$
  $H_a = \begin{pmatrix} \frac{dx}{\sqrt{r}} & \frac{dy}{\sqrt{r}} \\ -\frac{dy}{\sqrt{r}} & \frac{dx}{\sqrt{r}} \end{pmatrix}$ ▷ Compute Jacobian
  $Z_n = H_a C_n^m H_a^T + R_n$
  $H_p = \begin{pmatrix} -\frac{dx}{\sqrt{r}} & -\frac{dy}{\sqrt{r}} & 0 \\ \frac{dy}{r} & -\frac{dx}{r} & -1 \end{pmatrix}$
  $HZH[n] = H_p^T Z_n^{-1} H_p$ ▷ Shared Memory
  $\hat{\rho} = \sqrt{(\hat{x}_n - x_r)^2 + (\hat{y}_n - y_r)^2}$
  $\hat{\beta} = \arctan(\frac{\hat{y}_n - y_r}{\hat{x}_n - x_r})$
  $\delta = \begin{pmatrix} \sqrt{r} - \hat{\rho} \\ angle - \hat{\beta} \end{pmatrix}$
  threadSynch();
end for

Algorithm A15 Σ_n computation

for $n = threadId$ to N_c, step = $blockDim$ **do**
  if $n = 0$ **then**
    $\Sigma_{nminus1} = \Sigma_0^m$
  else
    $\Sigma_{nminus1} = Sigmas[n]$
    $\Sigma_n^{-1} = HZH[n] + \Sigma_{nminus1}^{-1}$
    $Sigmas[n] = \Sigma_n$ ▷ Shared Memory
  end if
  threadSynch();
end for

Algorithm A16 μ correction computation

for $n = threadId$ to N_c, step = $blockDim$ **do**
  $SHZ = \Sigma_n H_p^T Z_n^{-1} \delta$
  $\mu_0[n] = SHZ[0]$ ▷ Shared Memory
  $\mu_1[n] = SHZ[1]$ ▷ Shared Memory
  $\mu_2[n] = SHZ[2]$ ▷ Shared Memory
  threadSynch();
end for

Algorithm A17 Correction Application

if $threadId == 0$ **then**
    $x_t^m = \mu_0[N_c - 1]$
    $y_t^m = \mu_1[N_c - 1]$
    $\theta_t^m = \mu_2[N_c - 1]$
end if

Appendix A.6. Landmark Estimation

Algorithm A18 Landmark Initialization

$label = z_n[3]$
$particleSeenLandmarks[label] = \text{True}$
$reinitW$ = True ▷ Initialize the weights of the particles
$angle = z_n[0] + \theta_r^m$
$x_n^m = x_r^m + z_n[0] \cdot \cos(angle)$ ▷ Initialize Landmark Position
$y_n^m = y_r^m + z_n[1] \cdot \sin(angle)$

$dx = x_n^m - x_r^m$ ▷ position of the landmark w.r.t. particle pose
$dy = y_n^m - y_r^m$
$\hat{\rho}^2 = dx^2 + dy^2$
$\hat{\rho} = \sqrt{\hat{\rho}^2}$ ▷ Expected Range
$H = \begin{pmatrix} \frac{dx}{\hat{\rho}} & \frac{dy}{\hat{\rho}} \\ -\frac{dy}{\hat{\rho}^2} & \frac{dx}{\hat{\rho}^2} \end{pmatrix}$ ▷ Compute Jacobian
$C_n^m = H^{-1} \cdot Q \cdot {H^{-1}}^T$

Algorithm A19 Landmark Update

$dx = x_n^m - x_r^m$ ▷ position of the landmark w.r.t. particle pose
$dy = y_n^m - y_r^m$
$\hat{\rho}^2 = dx^2 + dy^2$
$\hat{\rho} = \sqrt{\hat{\rho}^2}$ ▷ Expected Range
$angle = \arctan(\frac{dy}{dx})$
$\hat{\beta} = angle - \theta_r^m$ ▷ Expected Bearing
$H = \begin{pmatrix} \frac{dx}{\hat{\rho}} & \frac{dy}{\hat{\rho}} \\ -\frac{dy}{\hat{\rho}^2} & \frac{dx}{\hat{\rho}^2} \end{pmatrix}$ ▷ Compute Jacobian
$K = C_n^m \cdot H^T \cdot (H \cdot C_n^m \cdot H^T + Q)^{-1}$ ▷ Kalman Gain
$d\rho = z_n[0] - \hat{\rho}$ ▷ Innovation
$d\beta = z_n[0] - \hat{\beta}$
$x_n^m = K[0] \cdot d\hat{\rho} + K[1] \cdot d\hat{\beta}$ ▷ Update Landmark Position
$y_n^m = K[2] \cdot d\hat{\rho} + K[3] \cdot d\hat{\beta}$
$C_n^m = (I - K_n^m H_a) C_n^m$ ▷ Update Landmark Covariance
$den =$det$(2\pi \cdot Q)$
$W_c[n] = \frac{1}{den^2} \cdot \begin{pmatrix} d\rho & d\beta \end{pmatrix} \cdot H^{-1} \cdot \begin{pmatrix} d\rho \\ d\beta \end{pmatrix}$ ▷ Compute local weight

Algorithm A20 Particle Weight Update

if $reinitW$ **then**
    $w_c = 1/N_p$
else
    $w_c =$ atomicMultiplication(W_c)
end if

References

1. Wang, K.; Zhao, G.; Lu, J. A Deep Analysis of Visual SLAM Methods for Highly Automated and Autonomous Vehicles in Complex Urban Environment. *IEEE Trans. Intell. Transp. Syst.* **2024** , *25*, 10524–10541. [CrossRef]
2. Zhuang, L.; Zhong, X.; Xu, L.; Tian, C.; Yu, W. Visual SLAM for Unmanned Aerial Vehicles: Localization and Perception. *Sensors* **2024**, *24*, 2980. [CrossRef]
3. Ding, S.; Zhang, T.; Lei, M.; Chai, H.; Jia, F. Robust visual-based localization and mapping for underwater vehicles: A survey. *Ocean. Eng.* **2024**, *312*, 119274. [CrossRef]
4. Zhang, Z.; Cheng, Y.; Bu, L.; Ye, J. Rapid SLAM Method for Star Surface Rover in Unstructured Space Environments. *Aerospace* **2024**, *11*, 768. [CrossRef]
5. Singh, J.; Tyagi, N.; Singh, S.; Ali, F.; Kwak, D. A Systematic Review of Contemporary Indoor Positioning Systems: Taxonomy, Techniques, and Algorithms. *IEEE Internet Things J.* **2024**, *11*, 34717–34733. [CrossRef]
6. Yue, X.; Zhang, Y.; Chen, J.; Chen, J.; Zhou, X.; He, M. LiDAR-based SLAM for robotic mapping: State of the art and new frontiers. *Ind. Robot. Int. J. Robot. Res. Appl.* **2024**, *51*, 196–205. [CrossRef]
7. Wang, H.; Li, M. A New Era of Indoor Scene Reconstruction: A Survey. *IEEE Access* **2024**, *12*, 110160–110192. [CrossRef]
8. Zhu, J.; Li, H.; Zhang, T. Camera, LiDAR, and IMU based multi-sensor fusion SLAM: A survey. *Tsinghua Sci. Technol.* **2023**, *29*, 415–429. [CrossRef]
9. Deng, W.; Dong, Z.; Zhang, L. Single-sensor-based and multi-sensor fusion SLAM: A survey. In Proceedings of the Ninth International Symposium on Sensors, Mechatronics, and Automation System (ISSMAS 2023), Xiamen, China, 6–8 January 2023; SPIE: Pamiers, France, 2024; Volume 12981, pp. 186–194.
10. Motlagh, H.D.K.; Lotfi, F.; Taghirad, H.D.; Germi, S.B. Position Estimation for Drones based on Visual SLAM and IMU in GPS-denied Environment. In Proceedings of the 2019 7th International Conference on Robotics and Mechatronics (ICRoM), Tehran, Iran, 20–21 November 2019 ; pp. 120–124. [CrossRef]
11. De Pace, F.; Kaufmann, H. A systematic evaluation of an RTK-GPS device for wearable augmented reality. *Virtual Real.* **2023**, *27*, 3165–3179. [CrossRef]
12. Bresson, G.; Alsayed, Z.; Yu, L.; Glaser, S. Simultaneous localization and mapping: A survey of current trends in autonomous driving. *IEEE Trans. Intell. Veh.* **2017**, *2*, 194–220. [CrossRef]
13. Zheng, S.; Wang, J.; Rizos, C.; Ding, W.; El-Mowafy, A. Simultaneous Localization and Mapping (SLAM) for Autonomous Driving: Concept and Analysis. *Remote Sens.* **2023**, *15*, 1156. [CrossRef]
14. Chen, Z. Bayesian filtering: From Kalman filters to particle filters, and beyond. *Statistics* **2003**, *182*, 1–69. [CrossRef]
15. Kalman, R.E. A New Approach to Linear Filtering and Prediction Problems. *J. Basic Eng.* **1960**, *82*, 35–45. [CrossRef]
16. Kalman, R.E.; Bucy, R.S. New results in linear filtering and prediction theory. *J. Basic Eng.* **1961**, *83*, 95–108. [CrossRef]
17. Julier, S.J.; Uhlmann, J.K. New extension of the Kalman filter to nonlinear systems. In *Signal Processing, Sensor Fusion, and Target Recognition VI*; SPIE: Pamiers, France, 1997; Volume 3068, pp. 182–193.
18. Maybeck, P.S. *Stochastic Models, Estimation and Control*; Elsevier: Amsterdam, The Netherlands, 1982.
19. Dellaert, F.; Fox, D.; Burgard, W.; Thrun, S. Monte carlo localization for mobile robots. In Proceedings of the 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C), Detroit, MI, USA, 10–15 May 1999; Volume 2, pp. 1322–1328.
20. Montemerlo, M. FastSLAM: A factored solution to the simultaneous localization and mapping problem. *AAAI/IAAI* **2002**, *593598*.
21. Montemerlo, M.; Thrun, S.; Koller, D.; Wegbreit, B. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *IJCAI* **2003**, *3*, 1151–1156.
22. Thrun, S.; Burgard, W.; Fox, D. Probalistic robotics. *Kybernetes* **2006**, *35*, 1299–1300. [CrossRef]
23. Wurm, K.M.; Stachniss, C.; Grisetti, G. Bridging the gap between feature-and grid-based SLAM. *Robot. Auton. Syst.* **2010**, *58*, 140–148. [CrossRef]
24. Andersone, I. Heterogeneous map merging: State of the art. *Robotics* **2019**, *8*, 74. [CrossRef]
25. Strasdat, H.; Montiel, J.; Davison, A.J. Real-time monocular SLAM: Why filter? In Proceedings of the 2010 IEEE International Conference on Robotics and Automation, Anchorage, AK, USA, 3–7 May 2010; pp. 2657–2664.
26. Aulinas, J.; Petillot, Y.; Salvi, J.; Lladó, X. The SLAM problem: A survey. *Artif. Intell. Res. Dev.* **2008**, 363–371 .
27. Bavle, H.; Sanchez-Lopez, J.L.; Cimarelli, C.; Tourani, A.; Voos, H. From slam to situational awareness: Challenges and survey. *Sensors* **2023**, *23*, 4849. [CrossRef] [PubMed]
28. Cadena, C.; Carlone, L.; Carrillo, H.; Latif, Y.; Scaramuzza, D.; Neira, J.; Reid, I.; Leonard, J.J. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Trans. Robot.* **2016**, *32*, 1309–1332. [CrossRef]
29. Picard, Q.; Chevobbe, S.; Darouich, M.; Didier, J.Y. A survey on real-time 3D scene reconstruction with SLAM methods in embedded systems. *arXiv* **2023**, arXiv:2309.05349.
30. Yu, Y.; Zhu, K.; Yu, W. YG-SLAM: GPU-Accelerated RGBD-SLAM Using YOLOv5 in a Dynamic Environment. *Electronics* **2023**, *12*, 4377. [CrossRef]
31. Kumar, D.; Gopinath, S.; Dantu, K.; Ko, S.Y. JacobiGPU: GPU-Accelerated numerical differentiation for loop closure in visual SLAM. In Proceedings of the 2024 IEEE International Conference on Robotics and Automation (ICRA), Yokohama, Japan, 13–17 May 2024; pp. 1687–1693.

32. Muzzini, F.; Capodieci, N.; Cavicchioli, R.; Rouxel, B. Brief announcement: Optimized gpu-accelerated feature extraction for orb-slam systems. In Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, Orlando, FL, USA, 17–19 June 2023; pp. 299–302.
33. Hu, Z.; Fang, H.; Zhong, R.; Wei, S.; Xu, B.; Dou, L. GMP-SLAM: A real-time RGB-D SLAM in Dynamic Environments using GPU Dynamic Points Detection Method. *IFAC-Papersonline* **2023**, *56*, 5033–5040. [CrossRef]
34. Muzzini, F.; Capodieci, N.; Cavicchioli, R.; Rouxel, B. High-Performance Feature Extraction for GPU-Accelerated ORB-SLAMx. In Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), Valencia, Spain, 25–27 March 2024; pp. 1–2.
35. Kamburugamuve, S.; He, H.; Fox, G.; Crandall, D. Cloud-based parallel implementation of slam for mobile robots. In Proceedings of the International Conference on Internet of things and Cloud Computing, New York, NY, USA, 22–23 March 2016; pp. 1–7.
36. Lambertus, T.J.; Hobiger, T. Single point positioning by means of particle filtering on the GPU. In Proceedings of the 2019 European Navigation Conference (ENC), Warsaw, Poland, 9–12 April 2019; pp. 1–9.
37. Zhang, H.; Martin, F. CUDA accelerated robot localization and mapping. In Proceedings of the 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), Woburn, MA, USA, 22–23 April 2013; pp. 1–6.
38. Par, K.; Tosun, O. Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures. In Proceedings of the 2011 IEEE Intelligent Vehicles Symposium (IV), Baden-Baden, Germany, 5–9 June 2011; pp. 820–826.
39. Abouzahir, M.; Elouardi, A.; Bouaziz, S.; Latif, R.; Tajer, A. Large-scale monocular FastSLAM2. 0 acceleration on an embedded heterogeneous architecture. *Eurasip J. Adv. Signal Process.* **2016**, *2016*, 88. [CrossRef]
40. Li, Q.; Rauschenbach, T.; Wenzel, A.; Mueller, F. EMB-SLAM: An embedded efficient implementation of rao-blackwellized particle filter based SLAM. In Proceedings of the 2018 3rd International Conference on Control, Robotics and Cybernetics (CRC), Penang, Malaysia, 26–28 September 2018; pp. 88–93.
41. Jia, S.; Yin, X.; Li, X. Mobile robot parallel PF-SLAM based on OpenMP. In Proceedings of the 2012 IEEE International Conference on Robotics and Biomimetics (ROBIO), Guangzhou, China, 11–14 December 2012; pp. 508–513.
42. Chao, M.A.; Chu, C.Y.; Chao, C.H.; Wu, A.Y. Efficient parallelized particle filter design on CUDA. In Proceedings of the 2010 IEEE Workshop On Signal Processing Systems, San Francisco, CA, USA, 6–8 October 2010; pp. 299–304.
43. Zhang, H.; Liu, Y.; Zhu, M.; Xiong, N.; Kim, T.h. An Effective FastSLAM Algorithm Based on CUDA. *Int. J. Grid Distrib. Comput.* **2016**, *9*, 143–158. [CrossRef]
44. Maskell, S.; Alun-Jones, B.; Macleod, M. A single instruction multiple data particle filter. In Proceedings of the 2006 IEEE Nonlinear Statistical Signal Processing Workshop, Cambridge, UK, 13–15 September 2006; pp. 51–54.
45. Abouzahir, M.; Elouardi, A.; Bouaziz, S.; Hammami, O.; Ali, I. High-level synthesis for FPGA design based-SLAM application. In Proceedings of the 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), Agadir, Morocco, 29 November–2 December 2016; pp. 1–8.
46. Abouzahir, M.; Elouardi, A.; Bouaziz, S.; Latif, R.; Tajer, A. FastSLAM 2.0 running on a low-cost embedded architecture. In Proceedings of the 2014 13th International Conference on Control Automation Robotics & Vision (ICARCV), Singapore, 10–12 December 2014; pp. 1421–1426.
47. Neira, J.; Tardós, J.D. Data association in stochastic mapping using the joint compatibility test. *IEEE Trans. Robot. Autom.* **2001**, *17*, 890–897. [CrossRef]
48. Rosten, E.; Porter, R.; Drummond, T. Faster and better: A machine learning approach to corner detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **2008**, *32*, 105–119. [CrossRef]
49. Montemerlo, M.; Thrun, S. Simultaneous localization and mapping with unknown data association using FastSLAM. In Proceedings of the 2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422), Taipei, Taiwan, 14–19 September 2003; Volume 2, pp. 1985–1991.
50. Yi, Y.; Huang, Y. Landmark sequence data association for simultaneous localization and mapping of robots. *Cybern. Inf. Technol.* **2014**, *14*, 86–95. [CrossRef]
51. Cooper, A.J. A Comparison of Data Association Techniques for Simultaneous Localization and Mapping. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA , 2005.
52. Nieto, J.; Guivant, J.; Nebot, E.; Thrun, S. Real time data association for FastSLAM. In Proceedings of the 2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422), Taipei, Taiwan, 14–19 September 2003; Volume 1, pp. 412–418.
53. Nicely, M.A.; Wells, B.E. Improved parallel resampling methods for particle filtering. *IEEE Access* **2019**, *7*, 47593–47604. [CrossRef]
54. Miguez, J. Analysis of parallelizable resampling algorithms for particle filtering. *Signal Process.* **2007**, *87*, 3155–3174. [CrossRef]
55. Gong, P.; Basciftci, Y.O.; Ozguner, F. A parallel resampling algorithm for particle filtering on shared-memory architectures. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012; pp. 1477–1483.
56. Gordon, N.J.; Salmond, D.J.; Smith, A.F.M. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEEE Proc. Radar Signal Process.* **1993**, *140*, 107–113. [CrossRef]
57. Efron, B.; Tibshirani, R.J. *An Introduction to the Bootstrap*; Chapman & Hall: London, UK, 1993.
58. Kitagawa, G. Monte-Carlo filter and smoother for non-Gaussian nonlinear state space models. *J. Comput. Graph. Stat.* **1996**, *1*, 1–25. [CrossRef]

59. Whitley, D. A genetic algorithm tutorial. *Stat. Comput.* **1994**, *4*, 65–85. [CrossRef]
60. Douc, R.; Cappe, O.; Moulines, E. Comparison of resampling schemes for particle filtering. In Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, Zagreb, Croatia, 15–17 September 2005; pp. 64–69.
61. Murray, L.M.; Lee, A.; Jacob, P.E. Parallel resampling in the particle filter. *J. Comput. Graph. Stat.* **2016**, *25*, 789–805. [CrossRef]
62. Dülger, Ö.; Oğuztüzün, H.; Demirekler, M. Memory coalescing implementation of Metropolis resampling on graphics processing unit. *J. Signal Process. Syst.* **2018**, *90*, 433–447. [CrossRef]
63. Klöckner, A.; Pinto, N.; Lee, Y.; Catanzaro, B.; Ivanov, P.; Fasih, A. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* **2012**, *38*, 157–174. [CrossRef]
64. Garland, M.; Le Grand, S.; Nickolls, J.; Anderson, J.; Hardwick, J.; Morton, S.; Phillips, E.; Zhang, Y.; Volkov, V. Parallel computing experiences with CUDA. *IEEE Micro* **2008**, *28*, 13–27. [CrossRef]
65. CUDA C Programming Guide. 2023. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide (accessed on 31 May 2023).
66. Dine, A.; Elouardi, A.; Vincke, B.; Bouaziz, S. Graph-based SLAM embedded implementation on low-cost architectures: A practical approach. In Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, USA, 26–30 May 2015; pp. 4612–4619.
67. Dine, A.; Elouardi, A.; Vincke, B.; Bouaziz, S. Speeding up graph-based SLAM algorithm: A GPU-based heterogeneous architecture study. In Proceedings of the 2015 IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Toronto, ON, Canada, 27–29 July 2015; pp. 72–73.
68. Vincke, B.; Elouardi, A.; Lambert, A. Real time simultaneous localization and mapping: Towards low-cost multiprocessor embedded systems. *Eurasip J. Embed. Syst.* **2012**, *2012*, 5. [CrossRef]
69. Blelloch, G.E. *Prefix Sums and Their Applications*; Technical Report CMU-CS-90-190; School of Computer Science, Carnegie Mellon University: Pittsburgh, PA, USA, 1990.
70. Blelloch, G.E. Scans as primitive parallel operations. *IEEE Trans. Comput.* **1989**, *38*, 1526–1538. [CrossRef]