GPU-Accelerated Numerical Differentiation for Loop Closure in Visual SLAM

by

Dhruv Kumar

B.Sc., University at Buffalo, 2018

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Dhruv Kumar**
Degree:	**Master of Science**
Thesis title:	**GPU-Accelerated Numerical Differentiation for Loop Closure in Visual SLAM**
Committee:	**Chair:** Nick Vincent Assistant Professor, Computing Science
	Steven Ko Supervisor Associate Professor, Computing Science
	Keval Vora Committee Member Associate Professor, Computing Science
	William N. Sumner Examiner Associate Professor, Computing Science

Abstract

This thesis introduces a technique that leverages a GPU to enhance the efficiency of loop closure in visual-inertial SLAM systems, particularly in the approximation of Jacobians using the Finite Difference Method (FDM). Traditional FDM techniques often encounter computational overhead due to repeated perturbations in pose graphs. This work addresses this challenge with a novel methodology that includes strategic graph partitioning and an optimized approach to Jacobian approximation. By integrating this technique into ORB-SLAM3's g2o framework, the linearization process is significantly improved in terms of speed and efficiency. The evaluation of this approach, conducted on 12 sequences of varying lengths from the EuRoC and TUM-VI datasets, demonstrates a speedup of up to 4.23x in the linearization stage and an overall performance improvement of up to 2.08x in the optimization process.

Keywords: GPU Optimization; Jacobian Linearization; SLAM Acceleration; Pose Graph Efficiency

Acknowledgements

I extend my heartfelt gratitude to Dr. Steven Ko for his invaluable guidance and support throughout this journey. Additionally, I am deeply thankful to Dr. Karthik Dantu from the University at Buffalo for sharing his expertise in SLAM and robotics with me. My experience at SFU was made memorable by the entire team at the Reliable Systems Lab (RSL); thank you all for the enriching experience. A special thanks goes to my family for their unwavering support. To my mom, who has always believed in me, your faith has been my driving force. I wish you could be here to share this moment. Lastly, I would like to express my sincere appreciation to the committee members for their time and valuable insights throughout the review process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Statement

This thesis introduces a new method that applies strategic hybrid graph partitioning to compute Jacobians using the Finite Difference Method (FDM) on GPUs, aimed at accelerating the linearization in nonlinear optimization for SLAM problems. By combining vertex-based partitioning for perturbations and edge-based partitioning for error computation, we improve the linearization process for loop closure in ORB-SLAM3's g2o framework.

Overview

Simultaneous Localization and Mapping (SLAM) [2, 10] is a fundamental technique in the field of robotics and autonomous systems. It addresses the critical challenge of how a robot can navigate an unknown environment without a pre-existing map. SLAM involves the dual tasks of localization, where a robot determines its position in space, and mapping, where the robot concurrently constructs a map of the environment. This process is crucial for a wide range of applications, from autonomous vehicles navigating city streets to robots exploring extraterrestrial surfaces. The essence of SLAM lies in its ability to solve these two tasks simultaneously – as the robot moves through an environment, it uses sensor data to build a map and uses features of this map to refine its understanding of its location.

This complex interplay of mapping and localization in SLAM, particularly in dynamic, real-world environments, demands significant computational resources. This is due to the need to continuously process and analyze large volumes of data from various sensors, and to update and refine the map of the environment in real time as changes occur. Modern SLAM systems [4, 32, 31, 13] employ a diverse array of sensors, including cameras, LiDAR (Light Detection and Ranging), and RGB-D (Red-Green-Blue-Depth) sensors, to capture environmental data. A notable example is ORB-SLAM3 [4], a state-of-the-art SLAM system known for its robustness and efficiency. It achieves visual-inertial SLAM by integrating data from cameras and inertial measurement units (IMUs). This integration enhances the system's ability to accurately track rapid movements and rotations, which is crucial in environments with varied and unpredictable dynamics. The collected data is processed to identify landmarks or features in the environment. These features are essential for creating a detailed map of the surroundings and for precisely estimating the robot's position within it. As the robot explores more of the environment, ORB-SLAM3 continuously updates and refines this map, leading to progressively more accurate localization. This advanced method of visual-inertial SLAM demonstrates how incorporating multiple data sources can significantly enhance the performance of SLAM systems in complex and changing environments.

The intricacies of SLAM involve two primary processes: visual odometry and loop closure. Visual odometry is pivotal in estimating a robot's motion based on sequential image data from cameras, which often capture footage at high rates such as 30 frames per second. This process is complex, involving the extraction and tracking of visual features across consecutive frames to infer movement, as demonstrated in Figure 1.1. Loop closure is a critical yet computationally demanding component of SLAM. It addresses the cumulative error, or 'drift', that accumulates over time in the robot's estimated trajectory and the map it constructs. This drift is inevitable due to minor inaccuracies in sensor readings and visual odometry estimations. The loop closure module identifies when the robot revisits a previously mapped area and uses this information to minimize the difference between the actual path and the perceived path, aligning the current map with past observations.

Given the high frequency of data input, especially from advanced sensors like 30fps cameras and 10Hz LiDARs, processing this information in real-time is a substantial challenge. In visual SLAM, ensuring the loop closure operates efficiently and accurately is crucial, especially in applications requiring quick, responsive navigation like autonomous vehicles or drones.

In several such systems, the mapping component is efficiently represented through a pose graph. This graph structure serves as a spatial framework, capturing the essential relationships within an environment [14]. Within this framework, the robot's various positions (or 'poses') are denoted as vertices, and the measurements from odometry - which provide relative motion information - act as edge constraints between these vertices. Examples are depicted in Figure 1.2. These constraints play a crucial role in maintaining the accuracy of the robot's understanding of its environment.

A key challenge in this process arises during loop closure, a scenario where the robot revisits a previously mapped area as mentioned above. Addressing this challenge is critical, as it helps in reconciling the constraints and minimizing any discrepancies in the overall map. Essentially, it is an exercise in recalibrating the map, aligning the robot's past positions with its current location, thereby ensuring both continuity and accuracy in the representation of the environment. This task of realignment and error minimization in the pose graph is formulated as a Pose Graph Optimization (PGO) problem [6]. PGO is a sophisticated problem-solving framework, typically approached as a non-linear least squares minimization challenge. Iterative algorithms like the Levenberg-Marquardt (LM) method [28] are often employed to find optimal solutions to this complex problem.

One approach to simplify this non-linear optimization is by transforming it into a more manageable linear approximation, which can be achieved through methods like the Finite Difference Method (FDM). FDM is a numerical technique that approximates the derivative of a function using differences of the function values at discrete points [24]. In the context of PGO, FDM involves making small, calculated perturbations to pose dimensions – adjusted by a factor of δ – in both positive and negative directions. This process helps to gauge their impact on the cumulative error in the pose graph. While FDM offers valuable insights into the optimization process, it is inherently computationally demanding.

The primary challenge in using FDM for PGO lies in its computational intensity, which stems from the redundant computations required by repeated perturbations in scenarios involving constraints with identical poses. Each perturbation necessitates a recalculation, significantly increasing the computational load. These redundant computations lead to longer processing times and place greater demands on the hardware, particularly in scenarios where real-time processing is essential.

To enhance efficiency, libraries like g2o [15] expedite computation by implementing parallelization strategies, such as using APIs like OpenMP [7]. This parallelization is achieved by dividing the graph based on its edges. However, this method does not completely address the issue of redundant pose perturbations. The redundancy occurs when a pose in the graph is linked to multiple edges, and a single perturbation affects all these edges, leading to multiple perturbations overall.

To tackle the computational challenges of FDM in pose graph optimization, we introduce JacobiGPU, a novel method tailored for GPU (Graphics Processing Unit) utilization. Our approach involves partitioning the graph based on vertices and edges to optimize different computational tasks. The vertex-based partitioning ensures that each pose in the graph is perturbed only once, while the edge-based partitioning calculates the error for every constraint using both the perturbed and original results.

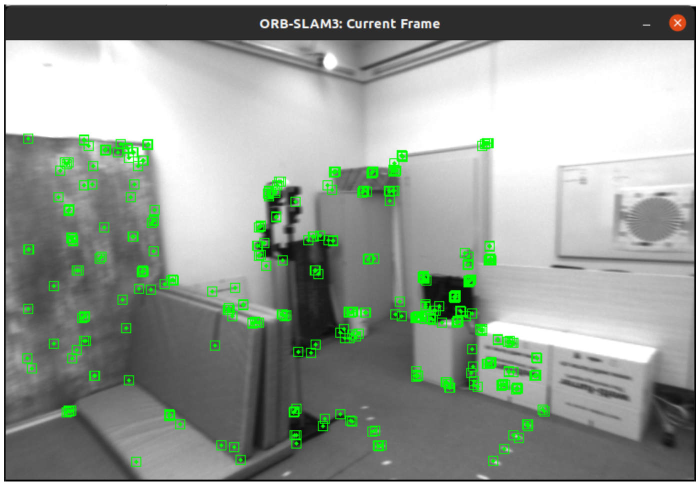
A significant benefit of using GPUs in this context is their capacity to handle large data volumes and perform multiple operations simultaneously, greatly decreasing processing times. JacobiGPU is designed to leverage these capabilities by optimizing GPU memory management, kernel operations, and the data transfer between the CPU and GPU. This strategy capitalizes on the strengths of GPUs to boost the overall efficiency of the optimization process.

We have seamlessly integrated JacobiGPU withing the g2o library, a core component of the ORB-SLAM3 system, focusing primarily on refining the linearization phase of the optimization process (Figure 4.1). This integration has been thoroughly tested using benchmark sequences from the EuRoC [3] and TUM-VI [33] datasets. JacobiGPU delivers up to a 4.23-fold increase in speed for the linearization step, and a 2.08-fold acceleration for the entire optimization process. In addition, we have compared the χ^2 error results from the pose graph optimization and the SLAM trajectory between the conventional g2o and our enhanced version incorporating JacobiGPU. These comparisons indicate that while JacobiGPU brings significant gains in terms of speed and efficiency, the integrity and reliability of the optimization process and the overall SLAM pipeline are maintained.

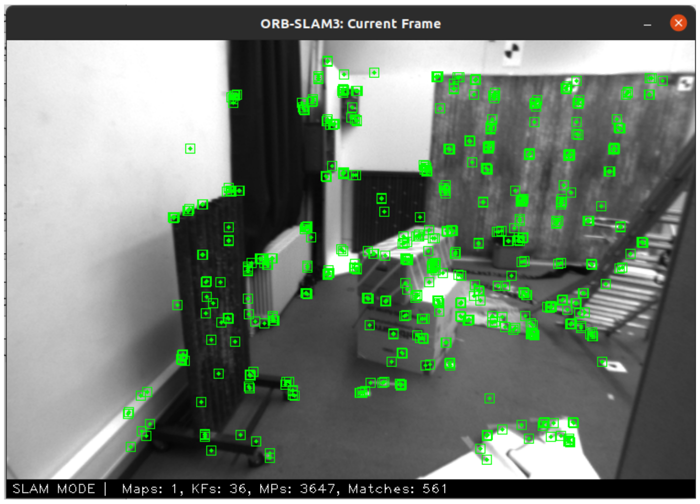
Note on Material Reuse

The research and findings presented in this thesis are primarily derived from work that will be presented in the conference paper titled "JacobiGPU: GPU-Accelerated Numerical Differentiation for Loop Closure in Visual SLAM" by Dhruv Kumar, Shishir Gopinath, Karthik Dantu, and Steven Y. Ko. This paper is accepted for presentation at the 2024 IEEE International Conference on Robotics and Automation (ICRA) © 2024 IEEE [25]. Portions of the text, along with some equations, figures and tables from this paper, are reused in this document.

IEEE Copyright Notice



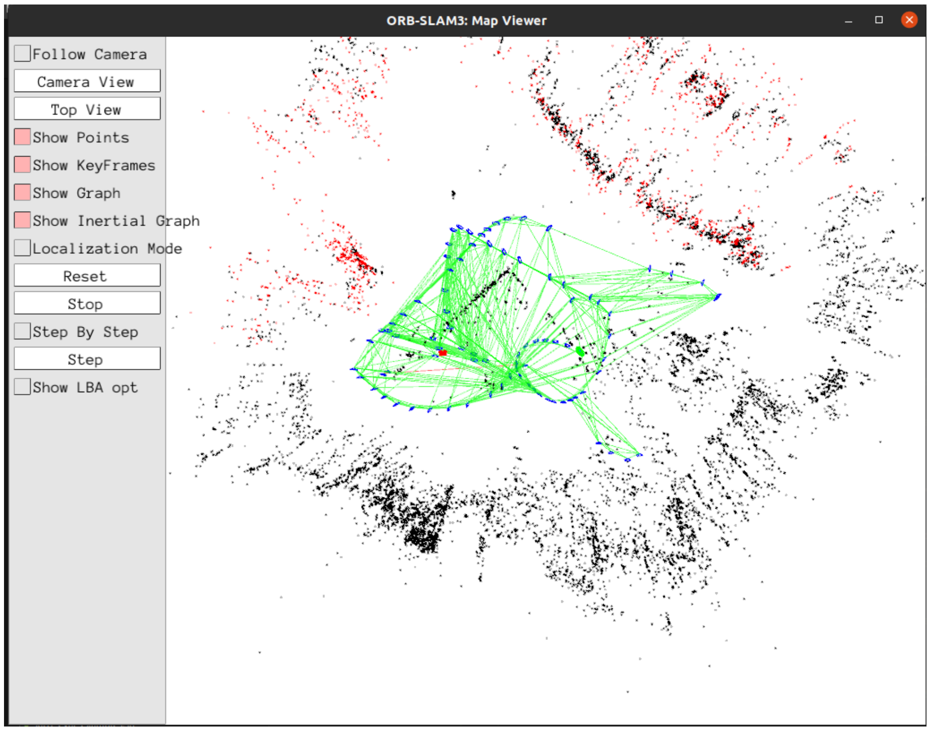
(a) V1O2



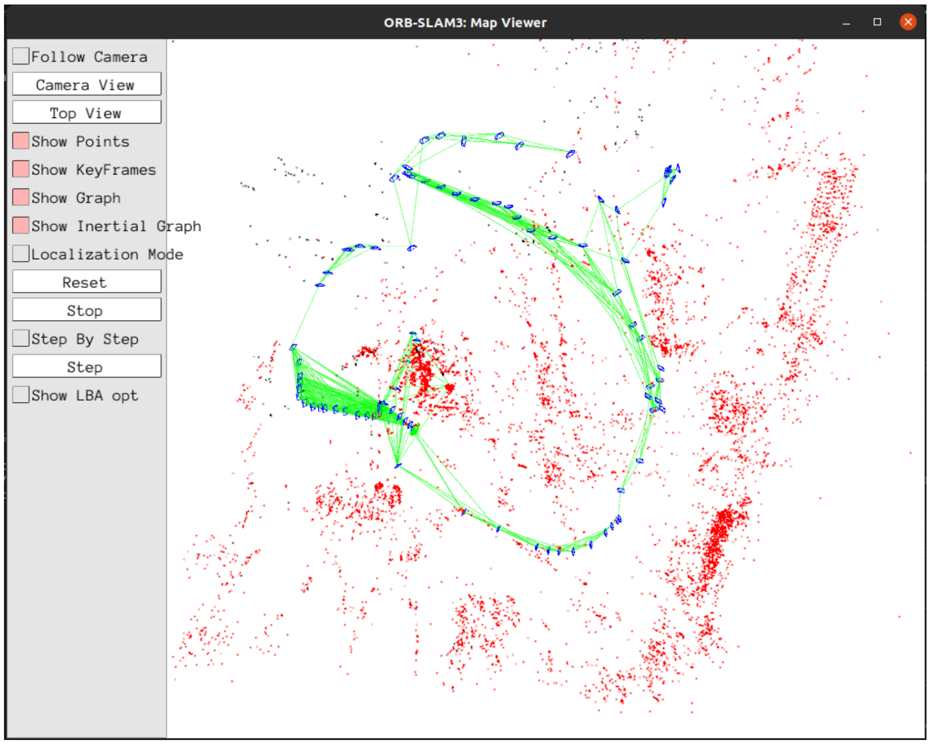
SLAM MODE | Maps: 1, KFs: 36, MPs: 3647, Matches: 561

(b) V201

Figure 1.1: Comparative frames from different sequences of the EuRoC dataset from ORB-SLAM3, depicting the nuanced process of feature extraction and mapping in SLAM across varied environmental complexities.



(a) V1O2



(b) V201

Figure 1.2: Visualization of pose graphs for the EuRoC dataset sequences V1O2 and V201 as processed by ORB-SLAM3. In these graphs, the green lines indicate the constraints or edges, while the blue marks represent the poses or nodes.

Chapter 2

Background

2.1 Pose Graph

A pose graph captures the spatial evolution of a robot as it traverses through an environment. It is a graphical representation where each node (or vertex) and edge encapsulates critical information about the robot's journey and interaction with its surroundings.

- **Vertices**: In a pose graph, vertices are the fundamental elements that represent the robot's poses at different points in time. A pose is a comprehensive term that combines the robot's position and orientation in a three-dimensional space, thus involving 6 Degrees of Freedom (DoF) - three for position (x, y, z) and three for rotation (roll, pitch, yaw). Each vertex in the graph is thus a mathematical construct, typically a combination of a 3D position vector and a 3×3 rotation matrix. These vertices are critical in capturing the trajectory of the robot, marking its path through the environment.

- **Edges**: The edges of the pose graph are equally significant. They represent the perceived movement or change between poses. These can be sequential, where each edge connects consecutive poses based on the robot's odometry data, forming a path that tracks the robot's immediate history. Alternatively, edges can manifest as loop closure constraints. These are more complex since they link a current pose with a previous one, usually identified with sophisticated place recognition algorithms. Loop closure edges are pivotal in SLAM as they help in correcting any drift or cumulative error that might have occurred in the robot's trajectory estimation over time.

While the construction of a pose graph provides a structured representation of a robot's trajectory and the environment, the true efficacy of this approach lies in the optimization of this graph. Pose graph optimization is a critical process where the goal is to refine the graph to best reflect the true layout of the environment and the most accurate path of the robot. This optimization process involves adjusting the vertices and edges of the graph to minimize the overall accumulated sensor error, effectively aligning the robot's perceived trajectory with the actual environment. The need for optimization arises because of the inherent inaccuracies in sensor data and the possibility of drift over time, especially in long trajectories or complex environments. The subsequent section on the pose graph optimization problem will talk more about the methodologies and algorithms used to achieve this optimization, discussing their roles, challenges, and the significance of accurately solving this problem in the context of effective SLAM.

2.2 Pose Graph Optimization

Pose Graph Optimization (PGO) is a key technique in refining the pose graph, ensuring that it accurately reflects the robot's movement and its interactions with the environment. The primary goal of PGO is to calibrate the vertices (representing the robot's poses) and their relational constraints (edges) in the graph to minimize the overall discrepancy between the estimated and actual paths. This process is essential for enhancing the precision and reliability of the SLAM system.

Mathematical Formulation of PGO: The optimization in PGO is formulated as a minimization problem, mathematically expressed as:

$$\min_{x} \sum_{\langle i,j \rangle \in G} e_{ij}^T \Omega_{ij} e_{ij}, \tag{2.1}$$

where G represents the set of all constraints within the pose graph, and Ω_{ij} is the information matrix corresponding to the constraint between poses i and j. This objective function seeks to minimize the sum of the squared errors across all constraints, leading to a more accurate and coherent mapping of the robot's trajectory.

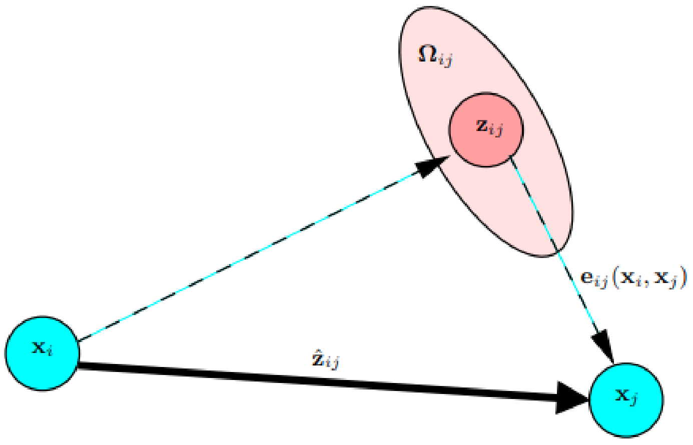


Figure 2.1: Visual representation of the error function $e_{ij}(x_i, x_j)$, illustrating the divergence between observed and predicted relative poses. Figure from [14].

Error Function in PGO: A critical component in PGO is the error function e_{ij}, which quantifies the divergence between the observed relative pose (or measurement) z_{ij} and the relative pose predicted from the current state estimates of the poses x_i and x_j. It is defined as:

$$e_{ij}(x_i, x_j) = z_{ij} - \hat{z}_{ij}(x_i, x_j), \tag{2.2}$$

where x_i and x_j denote the state variables for vertices i and j, and $\hat{z}_{ij}(x_i, x_j)$ represents the predicted relative pose based on the current estimates of x_i and x_j A visual representation of this error function, illustrating the divergence between observed and predicted relative poses, is provided in Figure 2.1, taken directly from [14]. The error function, e_{ij}, is instrumental in identifying and correcting inconsistencies within the pose graph.

Detailed Computation of Error Function: The error function e_{ij} comprises two main components: rotational E_R and translational E_t errors. These are computed as follows:

$$\begin{bmatrix} E_R \\ E_t \end{bmatrix} = \begin{bmatrix} \log_{\mathrm{SO}(3)}(R_w^i R_j^w R_i^j) \\ -R_w^i(R_j^w t_w^j) + t_w^i - t_j^i \end{bmatrix}, \tag{2.3}$$

where E_R and E_t represent the rotational and translational components of the error, respectively. The term $\log_{\mathrm{SO}(3)}(R_w^i R_j^w R_i^j)$ computes the rotational error E_R using the logarithm map of the special orthogonal group SO(3). This involves the multiplication of the relative rotation matrices R_w^i, R_j^w, and R_i^j, where:

- R_w^i is the rotation matrix from the world frame to the pose at vertex i.
- R_j^w is the rotation matrix from pose j back to the world frame.
- R_i^j is the measured or observed rotation from pose i to pose j.

The translational error E_t is calculated with the term $-R_w^i(R_j^w t_w^j) + t_w^i - t_j^i$, which represents the difference between the observed and estimated translations as mentioned above. This consists of:

- $-R_w^i(R_j^w t_w^j)$ computes the estimated translation from pose j to pose i, transformed into the coordinate frame of pose i.
- t_w^i and t_j^i represent the estimated and observed translations, respectively, with t_w^i being the translation from the world frame to pose i and t_j^i being the observed translation from pose j to pose i.

This detailed breakdown of the error function enables the optimization algorithm to accurately adjust the pose graph based on the differences between the estimated poses and observed measurements.

Optimization Algorithms: In Pose Graph Optimization, various algorithms are employed to iteratively refine the pose estimates. Common choices include the Gauss-Newton, Gradient Descent, and Levenberg-Marquardt algorithm [28]. Each of these algorithms has its own merits and is chosen based on the specific requirements and characteristics of the problem at hand.

The Levenberg-Marquardt algorithm is a particularly popular choice due to its robustness. It is a modification of the Gauss-Newton algorithm that interpolates between the Gauss-Newton and Gradient Descent methods [26, 34]. The update equation in the Levenberg-Marquardt algorithm is given by:

$$\Delta x = -(J^T \Omega J + \lambda \text{diag}(J^T \Omega J))^{-1} J^T \Omega e \tag{2.4}$$

In this equation:

- J represents the Jacobian matrix of all error functions. It quantifies how changes in the pose estimates affect the error.
- e is the vector of all errors, representing the discrepancies between the observed and estimated measurements.
- Ω is the block-diagonal matrix composed of information matrices for each measurement, reflecting the confidence or accuracy of these measurements.
- λ is the damping factor that is adjusted in each iteration. This factor is crucial in the algorithm's performance, as it balances the algorithm between the Gauss-Newton and Gradient Descent methods. A higher λ tends to make the algorithm behave more like Gradient Descent, making it more robust but possibly slower. Conversely, a lower λ shifts the behavior towards the Gauss-Newton method, which can be faster but less robust.

The choice of the damping factor λ and the adjustment strategy during iterations are key to the success of the Levenberg-Marquardt algorithm in PGO. Proper tuning of this parameter allows the algorithm to effectively navigate the complex error landscape of the pose graph, achieving convergence to an optimal solution.

2.3 Linearization

Linearization plays a key role in PGO, especially when dealing with complex, non-linear functions that describe the system's behavior. By employing linearization, we can approximate these non-linear functions with linear ones, making the problem more tractable.

Taylor Series Expansion: The essence of linearization in PGO lies in the use of the Taylor series expansion. For a vector-valued, multivariate function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$, we consider its behavior around a specific point $\mathbf{p}$ in the function's domain. The first-order

Taylor series expansion of $\mathbf{F}$ around $\mathbf{p}$ is given by:

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{p}) + \mathbf{J}(\mathbf{p})(\mathbf{x} - \mathbf{p}) \tag{2.5}$$

where $\mathbf{J}(\mathbf{p})$ represents the Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{p}$. This approximation simplifies the function by retaining only the first-order term, effectively turning the non-linear function into a linear one near the point $\mathbf{p}$.

In the context of PGO, linearization is vital. It allows the algorithm to handle complex relationships between pose adjustments and resulting errors in a more manageable way. The non-linear functions, which are inherently challenging to optimize directly, are approximated as linear ones close to the current estimate. This simplification is crucial for applying iterative optimization algorithms that require a linear model of the error function to update the pose estimates effectively. The Jacobian matrix $\mathbf{J}(\mathbf{p})$ in the Taylor series expansion is particularly important in this process. It quantifies how small changes in the pose parameters (represented by $\mathbf{x}$) around the point $\mathbf{p}$ affect the error function $\mathbf{F}$. Understanding the role and computation of the Jacobian matrix becomes essential as it directly influences how the PGO algorithm navigates the error landscape. This leads us to a deeper exploration of the Jacobian matrix in the following section, where we will discuss its structure, computation, and role in the optimization process in more detail.

2.4 The Jacobian Matrix

Jacobian Matrix captures the sensitivity of the error function to variations in the pose parameters, a key factor in the optimization process. It is a matrix of partial derivatives denoted as J in PGO. Each element J_{ij} within this matrix represents how the i^{th} component of the error function responds to infinitesimal changes in the j^{th} pose parameter. In essence, it captures the rate of change of each error component with respect to each pose parameter, thus guiding how the pose estimates should be updated to minimize the overall error.

Structure of the Jacobian Matrix: The Jacobian matrix is structured in a manner that each row corresponds to a different component of the error function and each column to a different pose parameter. It can be represented as:

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial x_1} & \frac{\partial e_1}{\partial x_2} & \cdots & \frac{\partial e_1}{\partial x_N} \\ \frac{\partial e_2}{\partial x_1} & \frac{\partial e_2}{\partial x_2} & \cdots & \frac{\partial e_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_M}{\partial x_1} & \frac{\partial e_M}{\partial x_2} & \cdots & \frac{\partial e_M}{\partial x_N} \end{bmatrix} \tag{2.6}$$

Here, $\frac{\partial e_i}{\partial x_j}$ is the partial derivative of the i^{th} error component with respect to the j^{th} pose parameter. The arrangement of this matrix is key to understanding how variations in each pose parameter affect different aspects of the error function, thereby informing the strategy for optimization. In our application, the structure of the Jacobian matrix is tailored to represent specific error components in its rows: for each column, indexed by i, the first three rows correspond to the derivatives of the rotational error with respect to a dimension which serves as a pose parameter represented by that column. Similarly, the following three rows are designated for the derivatives of the translational error components. This organization aligns with the six degrees of freedom in 3D pose estimation – three for rotation and three for translation – and is instrumental in directing the optimization process in PGO. This is computed using (2.3). The clarity in separating rotational and translational components in the Jacobian matrix allows for a nuanced adjustment of pose estimates, enhancing the precision and effectiveness of the SLAM algorithm.

In optimization algorithms, especially those like Levenberg-Marquardt which iteratively refine pose estimates, the Jacobian matrix J plays a pivotal role. It determines both the direction and magnitude of updates to pose parameters. The matrix's accuracy in reflecting how these adjustments impact the error is crucial for the efficiency and effectiveness of the optimization process, leading to more accurate and reliable pose graph estimation.

In summary, the Jacobian matrix J in PGO is key for gaining a understanding of the influence of pose parameter changes on the resultant error function. This understanding is essential for the successful application of optimization algorithms in SLAM. The Jacobian guides the iterative refinement of poses, which is vital for accurate and reliable mapping and localization. Moreover, it is instrumental in computing the update step Δx in iterative algorithms like Levenberg-Marquardt. The ability of the Jacobian matrix to precisely detail the relationship between pose adjustments and subsequent changes in error is critical for effective and efficient optimization. This precision aids in guiding the algorithm to an optimal solution, highlighting the importance of the Jacobian matrix in PGO.

Methods for Computing Derivatives: There are several methods for computing derivatives in the context of PGO, each with its own advantages:

- **Analytical Derivation**: This method involves computing the exact mathematical derivatives of the error functions. While precise, it can be complex and computationally intensive for non-linear functions.

- **Automatic Differentiation (Auto-Diff)**: Auto-Diff computes derivatives accurately and efficiently, using algorithms that can systematically apply the chain rule to computer programs.

- **Finite Difference Method (FDM)**: FDM offers a simpler, numerical approach to derivative estimation, which is particularly useful when analytical or auto-diff methods are impractical or overly complex.

As each method for computing derivatives presents unique advantages and drawbacks, the selection largely hinges on the specific nature and demands of the problem at hand. In the subsequent section, our focus shifts to the Finite Difference Method (FDM), where we will delve into its fundamental principles, its varied applications, and its particular significance within the realm of PGO. This discussion is especially pertinent to our work, which concentrates on tackling a crucial bottleneck in FDM and seeks to enhance its efficiency and effectiveness using GPUs.

2.5 Finite Difference Method

The Finite Difference Method (FDM) is a valuable numerical technique for the estimation of derivatives, especially in the computation of the Jacobian matrix. This technique proves particularly advantageous in scenarios where deriving an analytical or automatic differentiation (auto diff) Jacobian matrix is impractical, often due to the intricate nature of the functions encountered in PGO.

FDM operates by perturbing input values slightly and observing the resulting change in the function. This approach is based on the concept of approximating the derivative of a function at a point by analyzing its values at nearby points. The technique is most commonly applied using the central difference approximation, which provides a balance between accuracy and computational efficiency.

Central Difference Approximation: The central difference formula for approximating the derivative of a function with respect to a specific parameter is given by:

$$\frac{\partial F}{\partial x_{[j,k]}} \approx \frac{F(x_1, \dots, x_k + \delta, \dots) - F(x_1, \dots, x_k - \delta, \dots)}{2\delta}. \tag{2.7}$$

Here, $x_{[j,k]}$ represents the k^{th} dimension of the j^{th} pose, and δ is a small perturbation. The function F is evaluated at points slightly above and below the current estimate, and the difference in these values, divided by twice the perturbation δ, gives an approximation of the derivative.

Through the utilization of the central difference approximation in the FDM, incremental adjustments, or perturbations, are made to each pose parameter individually, while maintaining the other parameters constant across every constraint (edge) in the graph. This technique facilitates the approximation of derivatives for each component of the Jacobian matrix. In essence, the Jacobian matrix is incrementally constructed, with each column being formulated one by one, each aligning with a specific pose parameter. This systematic process allows for the compilation of a comprehensive and precise Jacobian matrix. By adopting FDM, PGO algorithms can more adeptly traverse the error landscape, enhancing their capability to refine pose estimations and bolster the overall precision of the SLAM process.

2.6 Lie Algebra and the Special Orthogonal Group SO(3)

Lie algebra plays a fundamental role in the mathematical underpinnings of various fields such as robotics, computer vision, and SLAM. It provides a formal framework for analyzing and representing the continuous symmetries of mathematical structures [17]. One of the key areas where Lie algebra finds application is in describing the properties and operations of rotation groups, particularly the Special Orthogonal Group SO(3).

Lie algebra, symbolized as $\mathfrak{g}$, forms an essential link with Lie groups, which represent groups capable of depicting continuous transformations [22]. The elements within these algebras are essentially the infinitesimal generators of such transformations, laying the groundwork for understanding complex movements in a structured manner. In the specific context of three-dimensional rotations, this algebra is represented as $\mathfrak{so}(3)$, comprising all skew-symmetric matrices [12]. These matrices are pivotal, encoding rotations around an axis through infinitesimally small angles and serving as the fundamental components for analyzing and calculating rotations within three-dimensional spaces.

Transitioning from the theoretical underpinnings provided by $\mathfrak{so}(3)$, we encounter the Special Orthogonal Group SO(3), a vital construct in the mathematical representation of rotations [16]. SO(3) encompasses all conceivable rotations in three-dimensional space, rendering it indispensable for any application that demands the manipulation of 3D orientations. It is formally defined as the group of 3x3 rotation matrices that are orthogonal, thus preserving vector lengths, and uniquely characterized by having a determinant of +1. This definition bridges the gap between the abstract algebraic formulations of $\mathfrak{so}(3)$ and the tangible application of rotations, illustrating the seamless integration of mathematical theory into the practical realm of three-dimensional orientation manipulation.

Two fundamental operations associated with SO(3) are the exponential and logarithmic maps, facilitating the conversion between the Lie algebra $\mathfrak{so}(3)$ and the Lie group SO(3). These operations are essential for efficiently handling 3D rotations:

- **Exponential Map** ($\mathrm{Exp} : \mathfrak{so}(3) \to \mathrm{SO}(3)$): Converts a rotation vector (axis-angle representation) into a rotation matrix using Rodrigues' rotation formula [8]. This transformation is key for composing rotations and transforming coordinates in 3D space.

$$\mathrm{ExpSO3}(\mathbf{w}) = \mathbf{I} + \sin(\theta)\hat{\omega} + (1 - \cos(\theta))\hat{\omega}^2, \tag{2.8}$$

  where $\hat{\omega}$ is the skew-symmetric matrix formed from the rotation vector $\mathbf{w}$, $\theta = \|\mathbf{w}\|$ is the magnitude of $\mathbf{w}$ denoting the rotation angle, and $\mathbf{I}$ represents the identity matrix. The skew-symmetric matrix $\hat{\omega}$ is defined as:

$$\hat{\omega} = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix}. \tag{2.9}$$

- **Logarithmic Map** (Log : SO(3) $\rightarrow \mathfrak{so}(3)$): This transformative operation reverts a rotation matrix back into a rotation vector, offering a streamlined representation of rotation that is ideal for processes like optimization and interpolation. The formula for this operation is articulated as follows:

$$\text{LogSO3}(\mathbf{R}) = \frac{\theta}{2\sin(\theta)} \begin{bmatrix} R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{bmatrix}, \tag{2.10}$$

where $\theta = \cos^{-1}\left(\frac{\text{trace}(\mathbf{R})-1}{2}\right)$ signifies the rotation angle. Here, trace($\mathbf{R}$), the sum of the diagonal elements of the rotation matrix $\mathbf{R}$, serves a pivotal role in determining θ. It encapsulates the cumulative effect of the rotation around the axis, directly influencing the computation of the rotation angle. The terms R_{ij} reference the matrix element at the intersection of the i^{th} row and j^{th} column, with R_{32}, R_{23}, and so on, specifically constructing a skew-symmetric matrix that encapsulates the axis of rotation through its off-diagonal components.

These transitions between $\mathfrak{so}(3)$ and SO(3) are fundamental in the articulation and manipulation of 3D rotations. By employing the exponential and logarithmic maps, we bridge the abstract algebraic domain of $\mathfrak{so}(3)$ with the tangible geometric realm of SO(3), thereby ensuring a precise and robust representation of rotations.

Chapter 3

Related Work

Graph-based approaches offer a sophisticated framework for accurately determining the position and orientation of robots while concurrently mapping their environment. Among the libraries developed for these crucial tasks, GTSAM [9] and g2o [15] stand out as widely used resources. Each brings unique capabilities to PGO, enhancing the efficiency and precision of SLAM operations.

GTSAM models the SLAM problem using factor graphs and Bayesian networks, offering a flexible and modular system for environment representation. This setup simplifies the integration of new sensors and information. A key feature, the iSAM algorithm [19], introduces an incremental optimization method for efficient real-time updates. By using a novel factorization strategy that recycles information from earlier computations, iSAM avoids the need for global optimization with each new data point, reducing computational demands. Its incremental approach ensures rapid solution delivery, ideal for the dynamic environments encountered in mobile robotics. This efficiency is especially valuable in extensive SLAM projects, allowing for the swift incorporation of new data without reprocessing the entire dataset. iSAM is adept at managing nonlinear optimization challenges typical in SLAM, through an advanced smoothing and mapping technique that effectively handles a growing dataset of measurements and variables.

Conversely, g2o serves as a versatile framework for optimizing graph-based models in both robotics and computer vision. Traditionally relying on the FDM for Jacobian calculations, it has recently incorporated the Ceres Solver's automatic differentiation (auto-diff) capabilities. This shift significantly enhances the precision and computational efficiency of gradient calculations, thereby improving the performance of PGO. This integration simplifies complex model implementations while boosting both the accuracy and efficiency of PGO. Nonetheless, despite its computational advantages, auto-diff's inherent limitations still exist as discussed in Section 3.2. g2o's flexibility is demonstrated through its application in diverse areas, including 3D reconstruction and multi-robot localization, highlighting its utility as a robust optimization tool.

While GTSAM and g2o both significantly contribute to the advancement and efficiency of SLAM operations through their unique features and optimizations, there remains room for enhancement, particularly in addressing the computational limitations associated with the FDM in g2o. This work aims to tackle one such limitation, focusing on refining the FDM's performance within the g2o framework to optimize gradient calculations further.

3.1 Pose Graph Optimization

The field of PGO has witnessed significant advancements, significantly enhancing the accuracy and efficiency of SLAM processes. Among these innovations, the work by Fan et al. [11] stands out by generalizing proximal methods for special Euclidean groups, achieving an impressive speedup in optimization processes. Their methodology, resulting in a 9x speedup compared to SE-Sync [30], represents a major advancement in optimization efficiency. This is particularly significant in light of the analysis by Juri et al. [18], which confirmed the superiority of SE-Sync over g2o in simulated environments [6]. Despite these advancements, the practical application of SE-Sync in real-world visual and visual-inertial sequences remains a fertile ground for further exploration, hinting at the potential for bridging the gap between theoretical optimization speeds and real-world applicability.

In parallel, the exploration of deep reinforcement learning (DRL) by Kourtzanidis et al. [23] marks a novel approach in PGO, particularly in addressing the longstanding challenge of escaping local minima in 2D graph optimizations. This venture into DRL not only underscores the adaptability and potential of machine learning in enhancing traditional optimization tasks but also sets a precedent for future explorations into AI-driven SLAM optimizations.

Furthermore, the innovative approaches proposed by Carlone et al. [5] eliminate the necessity for initial pose estimates, offering a path to overcoming one of the traditional hurdles in SLAM. By devising methods that circumvent the initial guess requirement, they showcase a possibility for achieving significant efficiency improvements over conventional iterative methods, such as the Levenberg-Marquardt algorithm. Despite the promising nature of these methods, their application has been primarily limited to 2D contexts, leaving an uncharted territory in the extension and validation of these techniques within 3D visual-inertial graph scenarios.

3.2 Automatic Differentiation

Automatic Differentiation (AD) is a computational technique within the realm of numerical optimization that transforming complex mathematical functions into a series of basic operations. By adeptly applying the chain rule of calculus, AD enables the precise and efficient computation of derivatives, a critical component in optimization algorithms. The utility of AD is exemplified in the work of Agarwal et al. [1], who have implemented AD within the

Ceres Solver to facilitate the linearization of constraints. In this framework, constraints are represented as residual blocks, each associated with a distinct cost function. These functions utilize a data structure known as a *Jet*, uniquely designed to support automatic differentiation within Ceres. The *Jet* structure efficiently computes not only the residuals but also the derivatives related to these residuals, thereby enhancing the solver's ability to iteratively refine solutions towards optimal states.

Extending the application of AD, Ren et al. [29] have developed SIMD-friendly *JetVectors* specifically designed for large-scale distributed GPU-based bundle adjustment tasks. With appropriate adjustments, this approach could similarly benefit PGO. This work demonstrates the adaptability of AD to modern parallel computing architectures, offering significant performance improvements in computationally demanding environments.

However, the adoption of AD is not without its challenges. A primary concern associated with AD is the memory overhead incurred during its operation. This issue arises as AD necessitates the retention of intermediate variables generated during the forward pass for use in the subsequent backward pass, thereby increasing the overall memory footprint of the computation. The utilization of dual numbers, as discussed by Pennestri et al. [27], exacerbates this memory demand. Dual numbers store both the value of a function and its derivative simultaneously, a requirement that, while beneficial for derivative computation, significantly impacts memory usage.

A further complication emerges in scenarios such as pose estimation, where the proximity of poses can introduce numerical instabilities into the computation. Agarwal et al. [1] highlights that numerical instability can arise when poses are close together or nearly overlap, as the computation of relative transformations may become unstable. This instability in the intermediate computations challenges the optimization process, potentially resulting in solutions that are not optimal. This underscores the need for careful management and validation of AD processes, especially in sensitive applications where precision and reliability are paramount.

In conclusion, while AD offers transformative potential for numerical optimization, its effective deployment requires mindful consideration of its computational and memory implications, as well as the inherent numerical challenges posed by specific application contexts.

Chapter 4

Approach

SLAM systems depend on sensor inputs to perform the dual tasks of understanding a device's location and mapping the environment around it. However, the real-world accuracy of sensor data is inherently limited, introducing errors that can significantly impact the effectiveness of SLAM operations. To mitigate these inaccuracies, processes such as bundle adjustment [34] and loop closure are employed, aiming to refine and minimize sensor errors through sophisticated optimization techniques.

The execution of these corrections, particularly during loop closure, presents a unique set of challenges. SLAM systems often need to momentarily halt their tracking functions to carry out these complex optimization procedures. The duration required to finalize this optimization is crucial; prolonged periods can compromise the accuracy and reliability of both mapping and localization efforts. This risk is magnified in environments characterized by sparse features, where every moment of inactivity can result in the loss of critical data. Thus, there is a pressing need to enhance the efficiency of these optimization stages to ensure valuable information is captured and preserved, maintaining the fidelity of the SLAM system's output despite the inherent limitations of sensor accuracy, which serves as the motivation for our work.

Loop closure primarily involves three crucial phases: building the graph, optimizing it, and adjusting the poses. Initially, pose graph information is compiled incrementally and stored. When a loop is detected, this information forms the foundation of the graph. Subsequently, the graph is optimized, at least in ORB-SLAM3 , to produce updated coordinates (x,y,z) for refining the poses. Following this, the system updates each pose accordingly. Among these stages, optimization emerges as the most time-consuming, as demonstrated in Table 4.1. This makes it the prime target for enhancing efficiency and the main focus of this study.

The optimization phase of loop closure consists of five key steps when employing the Levenberg-Marquardt algorithm as outlined below:

1. **Initialization:** This step prepares the optimization problem by setting initial estimates for the variables that will be optimized. This only happens in the first iteration.

Sequence	Construction (ms)	Optimization (ms)	Recovery (ms)
V102	1	16	6
V103	1	4	1
V202	1	18	5
V203	1	15	7
Room3	1	16	5
Room4	1	15	5
Magistrale1(1)	6	101	40
Magistrale1(2)	14	214	51
Magistrale2	8	139	32
Corridor1	13	156	51
Corridor5	9	133	36
Outdoors5	24	256	47
Outdoors7	35	619	100

Table 4.1: CPU times for core Loop Closure components across EuRoC and TUM-VI sequences in ORB-SLAM3.

2. **Computing Active Errors:** The algorithm calculates errors for the current state, identifying the differences between predicted outcomes and actual observations.

3. **Linearization:** In this phase, the algorithm approximates the behavior of the system around the current estimates using linear models. This is achieved by slightly adjusting the variables to see how changes impact the error, which is a crucial use of FDM.

4. **Determining Initial Lambda Value:** This involves selecting an initial value for the lambda parameter, which helps in balancing the algorithm's performance between gradient descent and Gauss-Newton methods.

5. **Employing a Solver:** The final step uses a solver to iteratively adjust the variables, aiming to minimize the overall error. The Levenberg-Marquardt algorithm excels in this step, striking a balance between convergence speed and accuracy.

The process of linearization, particularly when using FDM for approximating the Jacobian matrices, often emerges as the most time-consuming step in the optimization sequence, followed closely by the solver phase, as evidenced in Figure 5.1. With that in mind, our work focuses on migrating this computationally intensive task to GPU processing. This shift aims to significantly expedite the optimization process, enhancing the overall efficiency of loop closure, a pivotal improvement detailed in Figure 4.1.

The g2o framework enhances the efficiency of Jacobian computations by leveraging the parallel computing capabilities of OpenMP [7]. This approach distributes tasks across multiple threads by partitioning the graph into subgraphs based on edges, resulting in significant performance improvements. However, the primary cause of computational bottlenecks persists, mainly due to the redundant repetition of pose perturbations in densely interconnected

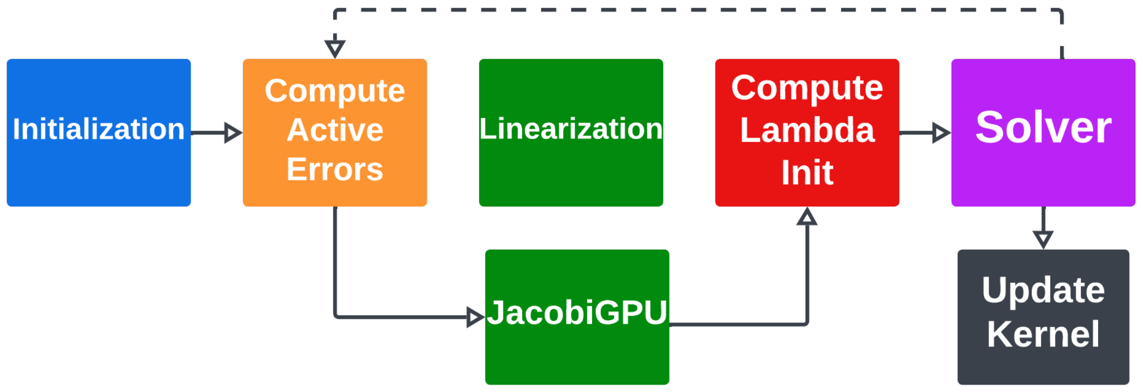


Figure 4.1: Visual Representation of the Levenberg-Marquardt Algorithm: Highlighting JacobiGPU's Role in Offloading the Linearization Step. Figure from [25].

pose graphs. Such redundancy stems from the graph's structure, often characterized by a higher number of edges than unique poses, leading to unnecessary recalculations as individual poses, central to overlapping observational areas, loop closures, and odometry-induced constraints, are repetitively perturbed through their associated edges.

For example, consider a pose, $P2$, linked to multiple constraints (e.g., landmarks, adjacent poses). Using FDM for Jacobian calculation, P_2 undergoes perturbation to estimate error gradients. For each of the 10 constraints connected to P_2, a separate perturbation is performed to calculate the respective errors:

- This results in redundant calculations as P_2 is perturbed 10 times independently for each constraint, despite the perturbations being identical.
- The redundancy could be avoided by sharing the perturbation results across constraints, thereby reducing the computational burden and increasing efficiency. This is the improvement our approach accomplishes.

GPUs are particularly well-suited to address this issue, as their architecture thrives when executing uniform tasks across many threads, such as the mentioned pose perturbations. Moreover, the graph data being represented as matrices and vectors, coupled with the primary operations being matrix-matrix or matrix-vector multiplications, aligns well with GPU strengths. This alignment is corroborated by the findings in [35], which highlight the GPU's superior performance in these types of computations. Additionally, GPUs provide a way to partition the graph in different ways at different points, which might also be possible on CPUs. However, the hybrid partitioning, coupled with the points mentioned earlier, has proven to be advantageous according to our findings.

Our solution, JacobiGPU, innovates by integrating specialized GPU kernels [21] into the FDM pipeline ensuring that each pose is perturbed precisely once per dimension. This enhancement directly addresses the inefficiencies previously identified, significantly streamlining the computation process and reducing the time needed for calculations. By reconfiguring the pipeline, JacobiGPU eliminates redundant recalculations, enhancing the optimization process's efficiency. This not only saves computational resources but also speeds up the adjustment of poses.

4.1 Overview

Our work is developed utilizing the SYCL framework [20], a high-level, cross-platform abstraction layer introduced by the Khronos Group. SYCL is designed to facilitate code execution across a diverse range of GPU architectures, thereby offering extensive compatibility and flexibility for application deployment on various hardware platforms. This capability ensures that the library can be versatile and adaptable to any GPU, including those from NVIDIA, AMD, or Intel, without necessitating specific coding tailored to each manufacturer's architecture.

Furthermore, the library is crafted to provide a user-friendly interface that abstracts the complexities of SYCL, thereby eliminating the need for applications integrating JacobiGPU to compile SYCL code directly. This abstraction simplifies the integration process, allowing developers to harness the power of SYCL and GPU acceleration effortlessly. As a result, developers can focus on leveraging the functionalities of the library, streamlining their development process.

Building on this foundation, JacobiGPU introduces two specialized kernels – *perturb* and *compute* – to efficiently tackle the challenge of redundant perturbations during the PGO linearization phase. The *perturb* kernel (k1), detailed in Section 4.3, partitions the graph based on vertices. It updates poses in both positive (+) and negative (-) directions concurrently, storing outcomes in designated intermediate buffers that hold the new and perturbed position of every pose in the graph (Figure 4.2). This ensures each pose is perturbed once, significantly enhancing efficiency by eliminating redundant computations.

Following the perturbation phase, the *compute* kernel (k2), outlined in Section 4.3, executes with its edge-based partitioning strategy. This allows for parallel error evaluations, ensuring updated pose estimates comply with the graph's constraints. Results, including derived Jacobian columns, are stored in the red buffer (Figure 4.2), ready for CPU utilization in solving the PGO problem.

The main insight upon which our work is built is the concept of hybrid partitioning based on the task at hand, rather than sticking to a simple single vertex or edge-based partitioning. This approach tailors the computational strategy to the specific needs of each phase of the graph optimization, leveraging the strengths of both vertex and edge-based methods to optimize performance and accuracy.

The development principles of JacobiGPU closely adhere to established best practices in GPU programming, as detailed in [21]. These best practices encompass several key areas:

1. **Memory Access Patterns:** JacobiGPU employs one-dimensional buffers to ensure uniform and predictable memory access, maximizing the GPU's throughput and reducing latency. This approach diverges from multi-dimensional array storage to optimize the GPU's data processing capabilities.
2. **Data Organization:** It groups similar data types in contiguous memory spaces to boost cache efficiency and spatial locality. This methodology aims to decrease cache misses and global memory accesses, enhancing performance and responsiveness.
3. **Host-Device Data Transfers:** To minimize latency from frequent data exchanges between CPU and GPU, JacobiGPU uses a third kernel for direct GPU pose updates post-solver, as shown in Figure 4.1. This reduces the need for constant data transfers, improving computation efficiency.
4. **Task Management:** JacobiGPU minimizes kernel launches and optimizes parallel processing capabilities by assigning specific tasks to each kernel. This strategy ensures efficient execution, leveraging the GPU's architecture for intensive computations with minimal delay.

The detailed implementation of these best practices and the impact on system performance are further elaborated in Section 4.2 and Section 4.3.

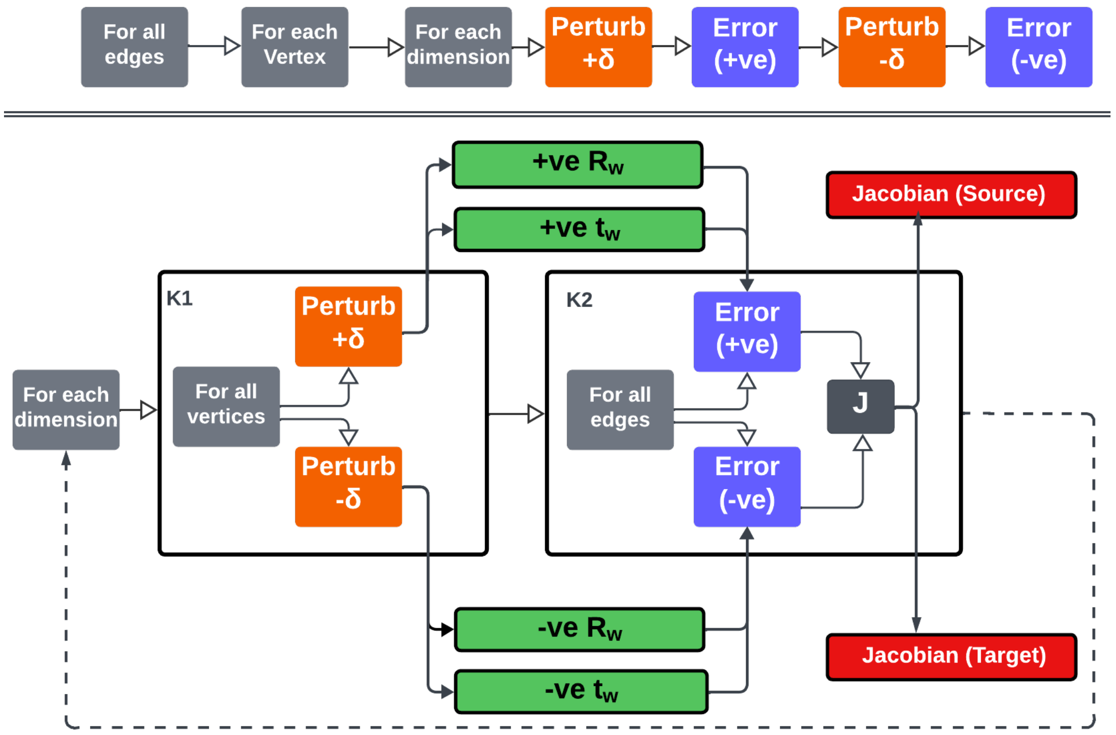


Figure 4.2: g2o (top) vs JacobiGPU (bottom) linearization overview. Box labeled 'J' symbolizes the computation of the Jacobian. Figure from [25].

4.2 Data Organization and Memory Architecture

GPUs are designed to maximize memory access efficiency through a hierarchical memory design characterized by varied access speeds. To enhance kernel performance, aligning with this architecture by adopting an appropriate data organization strategy is crucial. A notable technique involves reorganizing multi-dimensional data structures, such as rotation matrices and translation vectors for PGO, into one-dimensional (1-D) arrays or buffers. Such a reorganization facilitates linear and predictable data access patterns, enhancing memory throughput and kernel execution times due to optimized GPU bandwidth utilization.

Building on this, our work is designed to include 18 unique 1-D buffers that store the matrices and vectors in column-major format. Each buffer is designed to encapsulate specific components of the pose graph and the integrated multi-sensor system, aligning data management with the GPU's memory access paradigms.

Poses and Constraints

Pose and constraint information is distributed across ten 1-D buffers, divided as follows:

- **Constraint Data Buffers:** Two buffers are dedicated to storing the rotational and translational data corresponding to the graph's edges. They encapsulate the essential relative poses between frames or nodes, effectively representing the constraints that dictate spatial relationships and movements within the pose graph.
- **Pose Data Buffers:** Additionally, eight buffers are allocated for Pose Data, divided equally into four buffers for rotational matrices and four for translation vectors. This arrangement allows for a comprehensive depiction of:
    1. The orientation of the primary body relative to the global coordinate system (world), as detailed by the rotational matrices.
    2. The position of the primary body concerning the world, as outlined by the translation vectors.
    3. The orientation and position relationships between the primary body and any attached cameras, further elaborated by the supplementary buffers for rotation and translation.

This organization aims to optimize data storage by separating different types of data into distinct buffers, simplifying GPU data upload and enhancing cache efficiency. By leveraging spatial locality, where frequently accessed data elements are stored close together, this approach facilitates efficient data access across threads, streamlining the management of pose and constraint information while reducing the overhead associated with non-linear memory access patterns.

Intermediate Perturbed Pose

During the perturbation phase, essential for pose estimation, it's crucial to perform adjustments in both the positive (+) and negative (-) directions across each dimension. This bidirectional adjustment helps capture the effect of pose on error computation. A minimum of two buffers—one for translation and one for rotational data—is required for this operation. Adopting only two intermediate buffers would necessitate a sequential approach: first, perturbing in one direction and capturing the results with one kernel execution, followed by the computation of error through another kernel. This process would have to be replicated for perturbation in the opposite direction. For a system with four degrees of freedom (4DoF), such a method would result in sixteen kernel launches—eight for each direction of perturbation. For a six degrees of freedom (6DoF) system, this number escalates to twenty-four kernel launches:

$$\text{Kernel Launches} = 2\,(\text{for } +) \times \text{DoF} \times 2\,(\text{for } -) \tag{4.1}$$

However, by employing four intermediate buffers—two for storing the rotational and translational data from positive perturbations, and two for the data from negative perturbations—we can capture the results for both perturbation directions in a single kernel launch per dimension. This optimization effectively halves the number of required kernel executions:

$$\text{Optimized Kernel Launches} = \text{DoF} \times 2\,(\text{for both } + \text{ and } -) \tag{4.2}$$

This optimization lowers the number of kernel launches to eight for a 4DoF system and twelve for a 6DoF system with a slight uptick in memory usage. Let t be the time taken for one kernel launch to execute, n the number of kernel launches, m the memory usage per buffer, and b the number of buffers used. The memory used is then:

$$M = m \times b \tag{4.3}$$

The time saved by optimizing the number of kernel launches from 24 to 12 for a 6DoF system is given by:

$$T_{\text{saved}} = t \times (24 - 12) = 12t \tag{4.4}$$

The increase in memory usage from using two buffers to four buffers is:

$$\Delta M = m \times (4 - 2) = 2m \tag{4.5}$$

Jacobians

Within our pose graph optimization framework, two dedicated buffers are structured to hold the Jacobian matrices in column-major format, essential for the re-calibration of source and target poses associated with each graph edge. These buffers are organized to enhance the computational workflow:

- **Source Pose Jacobians Buffer:** This buffer is reserved for the Jacobian matrices pertaining to source poses, encapsulating the partial derivatives crucial for pose adjustments.
- **Target Pose Jacobians Buffer:** Similarly, this buffer is dedicated to the Jacobian matrices of target poses, containing the derivatives necessary for refining pose estimates.

The Compute Kernel, as mentioned in Section 4.3, is tasked with error computation, populating these buffers with Rotation and Translation errors. Each buffer is indexed by the constraint ID and the specific dimension under consideration, facilitating efficient data retrieval and updates:

$$\text{Buffer Size} = \text{Number of Constraints} \times 6 \times \text{DoF}, \tag{4.6}$$

where "DoF", again, represents the degrees of freedom within the system, which could be either 4 or 6, depending on the specific configuration being optimized. The factor of 6 accounts for the comprehensive data storage for a single dimension, or column, within the Jacobian matrix. Specifically, the arrangement accommodates both the rotational and translational error components, with the first three slots designated for rotational error and the subsequent three slots for translational error computed based on Equation 2.3. Further details on this configuration can be found in Section 2.4, which delves into the structure and significance of the Jacobian Matrix with respect to our work. This organization ensures that each dimension of the pose's degrees of freedom is fully represented in terms of its impact on both rotation and translation, facilitating detailed error analysis and subsequent pose adjustments.

Pose IDs

In SLAM systems, the assignment of IDs to keyframes – which represent specific poses – is often non-sequential, owing to the dynamic nature of updates with respect to pruning and the inclusion of new data . This lack of sequentiality disrupts the GPU's optimized patterns for sequential data access, a cornerstone for achieving high performance in computational tasks. The unpredictability introduced by non-sequential IDs complicates the GPU's ability to effectively prefetch data, which could lead to increased cache misses and, consequently, slower data retrieval times.

With this in mind, our work implements a systematic approach for reindexing keyframes as they are integrated into the system. This reindexing process assigns new, sequential IDs to each keyframe, starting from an ID of zero. The purpose of this strategy is twofold:

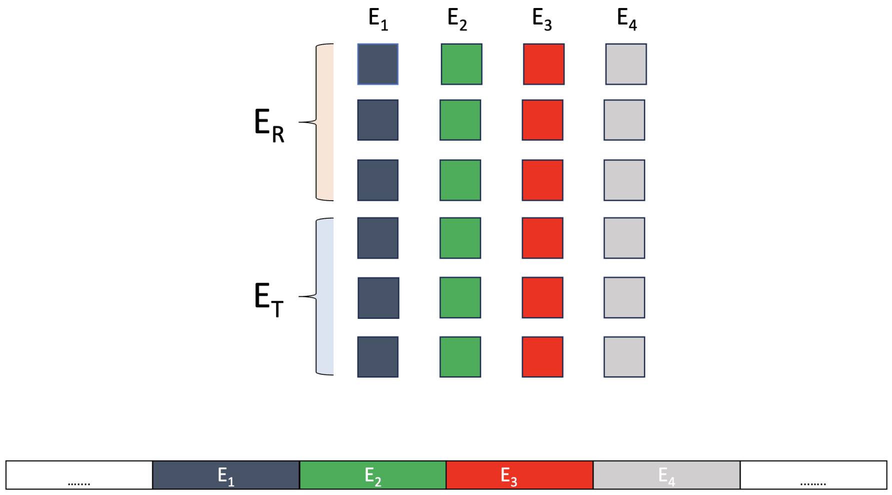


Figure 4.3: Illustration of converting a Jacobian matrix into 1-D column-major format for buffer storage in a 4DoF setup. E_n, E_r, and E_t denote error in the nth dimension, rotational, and translational errors, respectively. The bottom buffer demonstrates the column-major storage on GPU, with elements ordered from left to right, starting from the first row.

1. *Enhancing Data Access Efficiency:* By ensuring that keyframe IDs follow a sequential order, our library optimizes the GPU's data access patterns. Sequential IDs allow for more predictable data prefetching, thereby reducing cache misses and improving data retrieval speeds.

2. *Facilitating Data Organization:* The reorganized IDs are stored within two dedicated buffers. These buffers are structured to allow indexing based on the edge-ID, enabling swift and direct retrieval of pose data for both source and target poses associated with each graph constraint. This level of organization greatly simplifies the process of understanding and navigating the complex relationships between poses defined by the constraints in the pose graph.

Additionally on the host side, two critical mappings facilitate the seamless synchronization between CPU and GPU regarding pose IDs. One map translates pose IDs from the CPU context to their corresponding GPU context, while the reverse map translates pose IDs from GPU back to CPU. To streamline the interaction between these computing environments, a dedicated function is designed to input a pose and output its corresponding ID, accurately discerning whether the query is for the CPU or GPU context. This function achieves its objective by efficiently indexing the relevant map, providing a straightforward and rapid lookup mechanism. This setup significantly simplifies the integration process, ensuring that the transition of pose data across CPU and GPU contexts is both smooth and efficient.

4.3 Kernels

Perturb (k1)

The Perturb Kernel plays the role of perturbing each pose in the graph in designated dimensions. This subsection delves deeper into the kernel's functionality and its impact on the efficiency of the optimization process.

At its core, the Perturb Kernel operates by manipulating the original world-to-body transformation matrices, including translations associated with each pose. For each dimension, the kernel introduces small perturbations ($\delta\mathbf{R}$ for rotation and $\delta\mathbf{t}$ for translation) to simulate potential variations in the pose's orientation and position. This approach to pose adjustment is important as it facilitates a comprehensive exploration of the error landscape surrounding each estimated pose. The outcome of these perturbations, incorporating both positive and negative variations for each dimension, is systematically stored in green intermediate 1D buffers as show in Figure 4.2. These buffers serve as the groundwork for subsequent calculations in the optimization pipeline.

It does so by employing the central difference method, as per Equation 4.7, to compute the gradient of the error function efficiently. The kernel transforms the world-to-body relationship into a body-centric perspective, followed by applying the camera-to-body transformations. This allows for the calculation of a column within the Jacobian matrix, reflecting the sensitivity of the error function to changes in pose parameters.

$$\frac{\partial \mathcal{L}}{\partial x_{j,k}} \approx \frac{\mathcal{L}(R^{j,k}_{w_{\text{pos}}}, t^{j,k}_{w_{\text{pos}}}) - \mathcal{L}(R^{j,k}_{w_{\text{neg}}}, t^{j,k}_{w_{\text{neg}}})}{2\delta} \tag{4.7}$$

In this equation, $\mathcal{L}$ denotes the error function. The terms $R^{j,k}_{w_{\text{pos}}}$ and $t^{j,k}_{w_{\text{pos}}}$ represent the rotation and translation derived from a positive perturbation of pose j in the k^{th} dimension, while $R^{j,k}_{w_{\text{neg}}}$ and $t^{j,k}_{w_{\text{neg}}}$ correspond to those from the negative perturbation. This formulation underscores the significance of efficient pose transformations within the kernel, as they directly influence the computation of the gradient of the error function via the central difference method. Lie algebra [17] is used to facilitate the transformation between matrices and vector spaces, and vice versa.

The Perturb Kernel's capacity for processing multiple poses or nodes in parallel stands as one of its fundamental strengths, achieved through several key mechanisms. Firstly, it employs graph partitioning based on vertices. In this setup, each vertex (or pose) within the pose graph is treated as an independent entity for perturbation. These vertices are then processed concurrently, with different GPU threads handling different vertices. This approach allows for independent and simultaneous perturbation of each pose. This guarantees that each vertex is perturbed only once per dimension, avoiding the redundancy typical of sequential processing where a pose might be perturbed multiple times in the same dimension, corresponding to its connections or degree. Instead, this method allows for simultaneous perturbations and computations for each pose, ensuring efficiency and consistency in processing.

Compute (k2)

This kernel is launched after the Perturb kernel and is dependent on the output from that kernel. In the process of optimizing the pose graph, a critical step involves the accurate determination of the error between predicted and actual poses, followed by the estimation of the Jacobian matrix, which is essential for iterative optimization. This process is facilitated by this kernel, which leverages perturbations in pose parameters to compute these errors and subsequently estimate the Jacobian.

After introducing perturbations in each pose's parameters, the kernel proceeds to evaluate the error associated with these perturbed poses. The error calculation is grounded in the discrepancy between the predicted pose, as influenced by the perturbation, and the actual observed pose. This comparison is quantitatively expressed by Equation 2.3, which encompasses both rotational (E_R) and translational (E_t) components of the error. The kernel utilizes the difference in errors obtained from perturbations in both positive and negative directions to estimate the Jacobian matrix. As elucidated by Equation 4.7, this estimation involves computing the gradient of the error function with respect to each pose parameter: This differential approach effectively captures the sensitivity of the error function to variations in pose parameters, with each dimension's perturbation corresponding to a distinct column in the Jacobian matrix.

Following the estimation, the kernel integrates the computed Jacobian columns into the global Jacobian matrix. It identifies the vertices connected by the edges under consideration and places each pose's calculated column directly into the Jacobian buffers (Section 4.2). This systematic arrangement ensures that the optimization algorithm can accurately account for the influence of each pose on the overall error landscape, facilitating effective minimization of errors across the pose graph.

Update

This kernel has been developed to update poses directly on the GPU, based on 3D coordinate adjustments output by the solver at the end of each iteration. Given PGO's iterative nature, which traditionally would necessitate frequent data transfers between the CPU and GPU to adjust poses with each iteration's results, this update kernel performs updates directly where pose data reside—on the GPU. This approach avoids potential repetitive data transfers across iterations. The introduction of this kernel significantly improves the handling of iterative optimization algorithms by eliminating the inefficiencies associated with continuous data movement between the CPU and GPU. By maintaining pose data on the GPU and applying solver-derived adjustments directly, the need for data transfers between the CPU and GPU is reduced. As a result, the graph needs to be transferred only once for the entire optimization cycle, significantly streamlining the PGO workflow.

Chapter 5

Evaluation

5.1 Experimentation Setup

Our experimental setup features a 12th Gen Intel® Core™ i7-12700K CPU with 20 cores operating at 5.0 GHz, 62GB of RAM, and an NVIDIA GeForce RTX 3090 GPU, providing robust support for high-speed computations, handling large datasets, and performing intensive graphical processing tasks. In our framework, we adapt g2o for use within the ORB-SLAM3's optimization routines, initially enhancing performance through vectorization and the activation of OpenMP, which are not standard in the ORB-SLAM3 system. Vectorization enables the simultaneous processing of multiple data points, while OpenMP facilitates parallel computing, significantly reducing computational time. Whenever the CPU is mentioned in our results, it indicates that OpenMP was activated. Additionally, we compare our modified approach with g2o's parallel configuration to showcase the potential improvements when leveraging GPU capabilities.

For evaluation, we execute ORB-SLAM3 on sequences from the EuRoC and TUM-VI datasets, selected for their complex environments and varying loop closures in stereo-inertial setups. The EuRoC dataset, known for its challenging machine hall and Vicon room sequences, provides a test for aerial vehicle navigation, while the TUM-VI dataset offers diverse indoor and outdoor scenes essential for validating visual-inertial odometry. To validate our method, we compare the χ^2 error and trajectory accuracy between the original g2o and our modified version that incorporates our library. This comparison confirms that our library enhances loop closure optimization efficiency without negatively impacting the rest of the SLAM process.

Our benchmarks involve inputting sequences from the EuRoC and TUM-VI datasets, which contain varied environmental data and loop closure scenarios. These inputs are inherently nondeterministic because they represent real-world sensor data that can vary slightly with each run due to noise, sensor drift, and differing initial conditions. This nondeterminism means that our evaluations must account for variability in the results, necessitating multiple runs to obtain statistically significant outcomes. Therefore, we conduct ten repeated trials

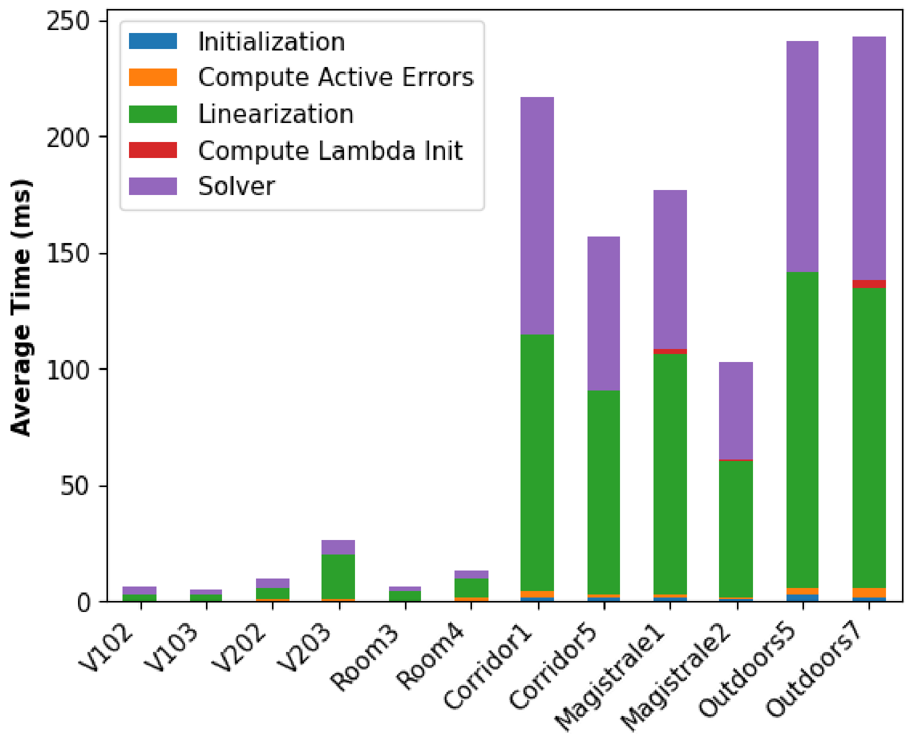


Figure 5.1: A breakdown of a Levenberg-Marquardt call for loop closure for EuRoC and TUM-VI sequences.

for each dataset sequence. This approach helps us confirm that improvements in performance metrics, such as reduced computational time, are consistent and reproducible under different conditions, rather than outcomes of random fluctuations in the data.

5.2 Runtime Performance

ORB-SLAM3's use of the Levenberg-Marquardt algorithm in g2o iteratively refines solutions, with Figure 5.1 detailing component timings. Linearization emerges as the most time-consuming step, followed by the solver, pinpointing our optimization efforts to enhance loop closure efficiency significantly. Integrating JacobiGPU show improvements in performance, particularly for larger graphs compared to smaller ones. Table 5.1 demonstrates up to a 4.23-fold increase in speed for Jacobian calculations and a 2.08-fold improvement in the overall optimization process. Notably, for the V103 dataset, where the graph size is smaller, our method underperforms slightly, attributed to the initial data transfer overhead outweighing computational gains.

Specifically focusing on linearization, Figure 5.2 provides a breakdown of the JacobiGPU kernel operations and the cumulative Jacobian data transfer time to the CPU over all iterations. Notably, the Perturb and Compute kernel's computation time for the V102 dataset

Sequence	**Graph Size**		**Linearization Time (ms)**		**Linearization**	**Total Optimization Time (ms)**		**Total Optimizer**
	Poses	**Constraints**	**CPU**	**JacobiGPU**	**Speedup (%)**	**CPU**	**JacobiGPU**	**Speedup (%)**
V102	91 ± 7	425 ± 35	5.63 ± 1.44	3.47 ± 0.85	+38.37 **(1.62x)**	9.08 ± 2.04	6.16 ± 2.34	+32.16 **(1.50x)**
V103	52 ± 9	425 ± 34	2.69 ± 1.30	3.12 ± 1.00	−15.99 **(0.86x)**	3.99 ± 1.70	4.16 ± 1.92	−4.26 **(0.95x)**
V202	57 ± 2	248 ± 13	5.26 ± 2.43	4.13 ± 1.86	+21.48 **(1.27x)**	7.88 ± 2.50	6.64 ± 1.02	+15.72 **(1.24x)**
V203	143 ± 41	566 ± 110	11.05 ± 0.82	6.79 ± 1.09	+38.46 **(1.63x)**	19.54 ± 1.77	12.57 ± 0.68	+35.63 **(1.55x)**
Room3	65 ± 4	502 ± 52	6.64 ± 2.16	2.65 ± 0.41	+60.12 **(2.50x)**	9.77 ± 3.07	4.84 ± 0.72	+50.46 **(2.02x)**
Room4	75 ± 7	874 ± 37	8.71 ± 2.42	4.19 ± 1.10	+51.90 **(2.08x)**	14.35 ± 1.29	8.10 ± 1.85	+43.55 **(1.77x)**
Magistrale1(1)	472 ± 36	3577 ± 199	56.43 ± 0.87	16.07 ± 0.53	+71.53 **(3.51x)**	92.75 ± 3.00	46.22 ± 0.90	+50.17 **(2.01x)**
Magistrale1(2)	1173 ± 205	9111 ± 1091	102.33 ± 1.94	25.11 ± 1.75	+75.47 **(4.08x)**	184.16 ± 13.42	95.86 ± 11.08	+47.94 **(1.92x)**
Magistrale2	543 ± 29	4637 ± 403	61.58 ± 1.69	14.56 ± 1.05	+76.35 **(4.23x)**	105.03 ± 3.73	50.47 ± 1.19	+51.95 **(2.08x)**
Corridor1	899 ± 45	7381 ± 731	93.56 ± 10.89	29.04 ± 31.08	+68.95 **(3.22x)**	177.58 ± 1.16	106.70 ± 17.40	+39.91 **(1.66x)**
Corridor5	843 ± 12	6442 ± 194	95.82 ± 6.89	29.73 ± 6.67	+68.98 **(3.22x)**	174.23 ± 19.04	106.04 ± 19.45	+39.13 **(1.64x)**
Outdoors5	1100 ± 30	11713 ± 392	135.66 ± 2.85	44.19 ± 2.27	+67.44 **(3.07x)**	222.89 ± 3.72	149.68 ± 3.07	+32.85 **(1.49x)**
Outdoors7	913 ± 20	9438 ± 397	129.79 ± 16.51	39.28 ± 0.85	+69.76 **(3.30x)**	223.89 ± 30.30	142.64 ± 25.46	+36.30 **(1.57x)**

Table 5.1: Linearization and total optimization times (in ms) for ORB-SLAM3 across 10 runs. The Magistrale1 sequence features two distinct loop closures, represented separately as Magistrale1(1) for the early stage and Magistrale1(2) for the later stage. The CPU Table from [25].

is around 0.5ms, while for the larger graph of outdoors5 (as shown in Table 5.1), the total computation time is 1.5ms. This is despite the graph for outdoors5 being nearly 11 times larger in terms of Poses and 25 times for constraints.

This section investigates how integrating JacobiGPU into the ORB-SLAM3 framework affects the computational efficiency during the linearization step of the Levenberg-Marquardt optimization algorithm. The measures used, component timing breakdowns, and speed improvements in Jacobian calculations and the overall optimization process are chosen to quantify the efficiency gains in the most computationally demanding aspects of the SLAM process directly.

5.3 GPU Memory Transfer

We used NVIDIA Nsight Systems for profiling memory transfers across different sequences, as illustrated in Figure 5.3. The 'Magistrale1' sequence emerged as the most demanding in terms of GPU memory transfer, with a peak transfer size of approximately 53.84MB. A substantial part of this transfer is attributed to host-to-device data movement, necessitated by the need to copy the entire graph's data to the GPU for processing. Specifically, the

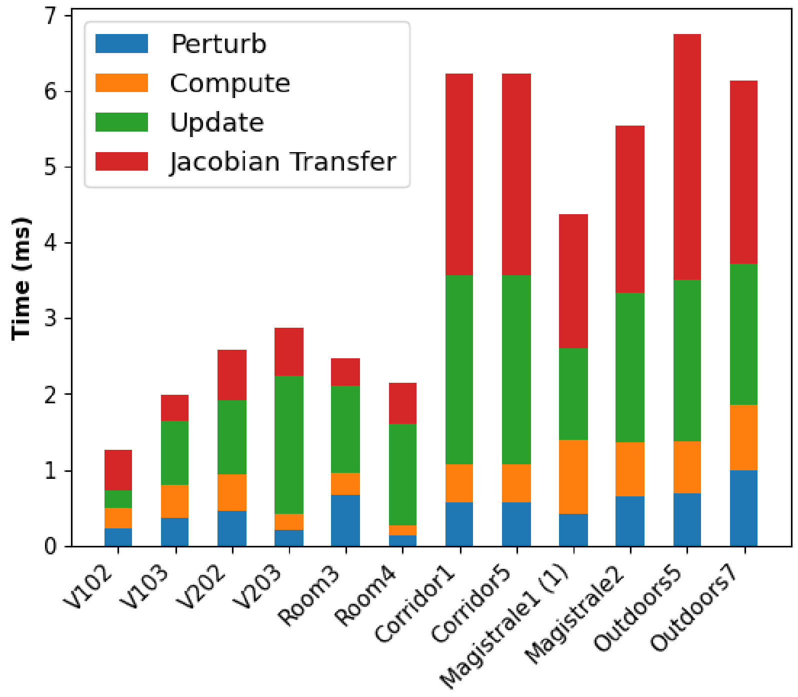


Figure 5.2: Breakdown of various components within JacobiGPU for EuRoC and TUM-VI sequences.

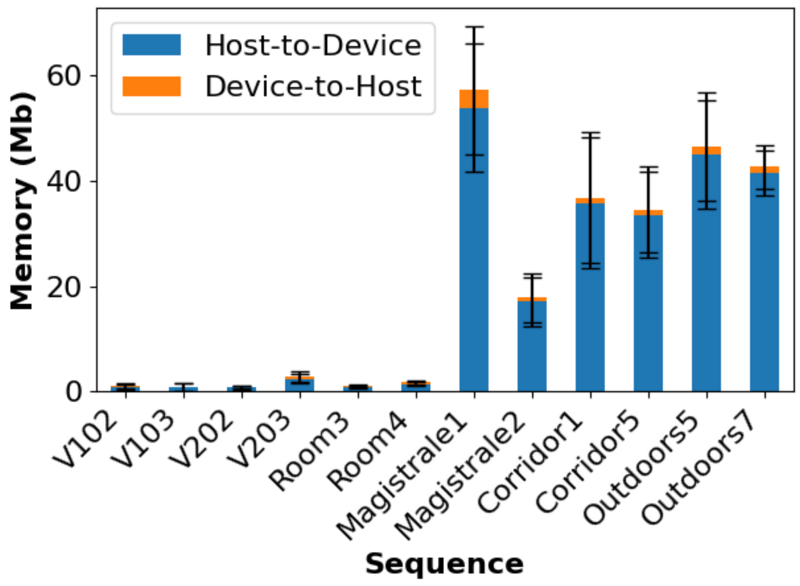


Figure 5.3: Average memory transfer for buffer allocation across sequences, as profiled by Nsight Systems over five runs. The ‘Magistrale1’ bar aggregates the memory transfers for both loop closures 1 and 2. Figure from [25].

transfer of Jacobian buffers from the device back to the host accounted for 3.4MB of the total transfer.

This subsection details the memory transfers between the CPU and GPU across various sequences, focusing on quantifying the total data transfer, especially the Jacobian buffers from GPU to CPU. This helps in understanding the correlation between the graph sizes and the volume of memory required for transfer, providing a clearer depiction of the system's operational demands.

5.4 Trajectory

To assess the impact of our modifications on trajectory accuracy, we conducted a comparative analysis by running the same sequence twice: first under the standard ORB-SLAM3 setup and then using our enhanced version. This process was repeated for all sequences in our study, with Figure 5.4 presenting the results for a representative sequence. Upon analyzing the trajectory data from both runs, we observed that the trajectories were similar overall. However, running the same sequences multiple times might lead to similar results due to the deterministic aspects of the algorithm or consistent sensor error patterns, such as repeatable processing of image features and consistent response to identical input data. It is important to consider these factors when evaluating the generalizability and improvements of the modified system in real-world conditions.

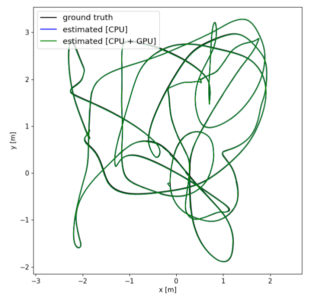
This subsection conducts a comparative analysis to ascertain whether the modified ORB-SLAM3 setup maintains trajectory accuracy relative to the standard configuration across identical datasets. This evaluation serves as an essential verification of the consistency and reliability of the system enhancements under real-world conditions.

5.5 Error Convergence

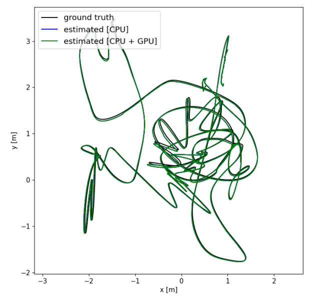
The χ^2 error metric, derived from the g2o optimization process, is pivotal for evaluating the accuracy and dependability of the optimization efforts within our SLAM framework. In comparing the outcomes from the standard ORB-SLAM3 setup with those from our modified implementation, we notice a similar error convergance. Detailed in Figure 5.5, the error profiles of both versions exhibit a perfect alignment, underscoring that our JacobiGPU-enhanced version maintains fidelity to the original implementation's behavior. This alignment not only validates the modifications we've introduced but also underscores the effectiveness of JacobiGPU in replicating the established optimization behavior without compromising on the system's reliability and accuracy. This finding demonstrates our ability to enhance computational efficiency while preserving the core functionalities and performance benchmarks of the original system.

This subsection investigates how the JacobiGPU-enhanced version of ORB-SLAM3 compares to the original in terms of χ^2 error convergence during the optimization process. The

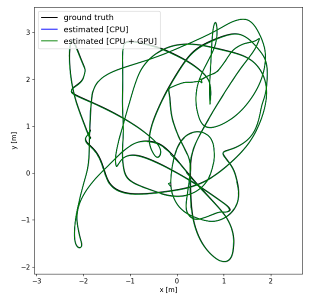
χ^2 error profiles from both versions are examined, ensuring that while computational efficiency is enhanced, the accuracy of the optimization is not compromised.



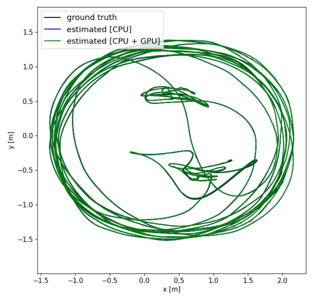
(a) V102



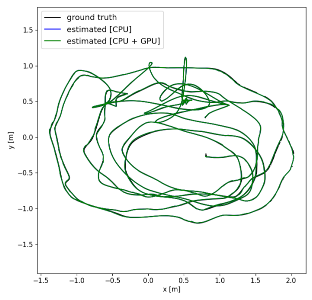
(b) V103



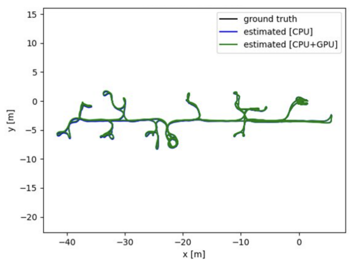
(c) V202



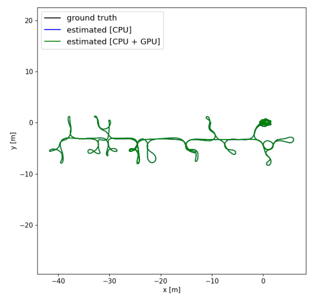
(d) Room3



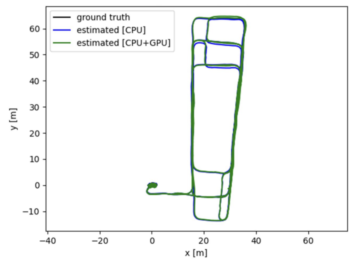
(e) Room4



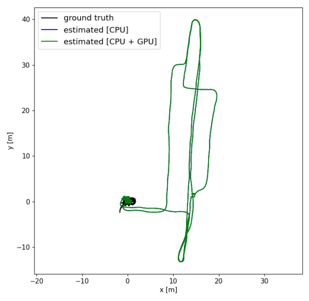
(f) Corridor1



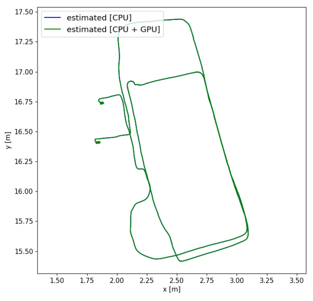
(g) Corridor5



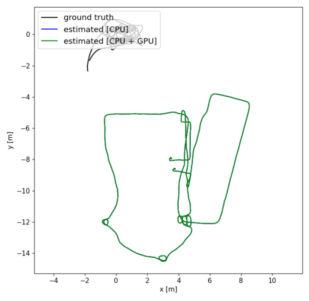
(h) Magistrale1



(i) Magistrale2

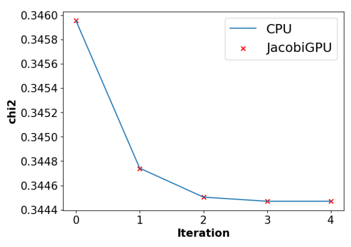


(j) Outdoors5

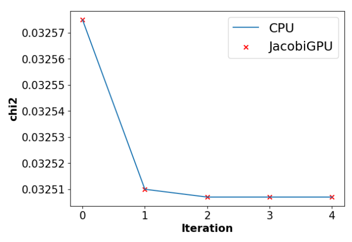


(k) Outdoors7

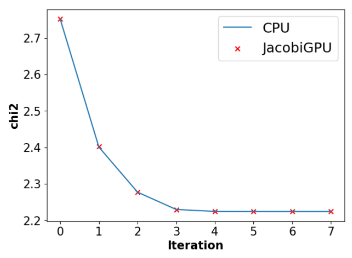
Figure 5.4: ORB-SLAM3 trajectories from two distinct runs of various sequences: the original CPU-based implementation (blue) versus the enhanced version with JacobiGPU (green). Some taken from [25].



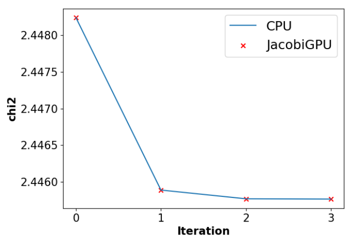
(a) V102



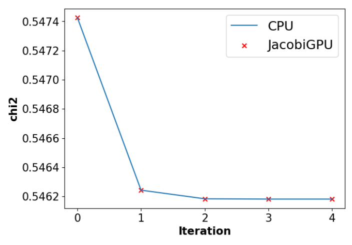
(b) V103



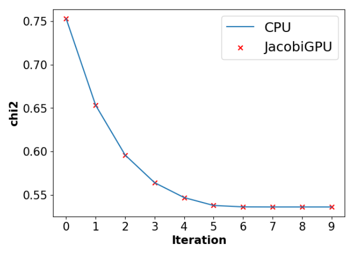
(c) V202



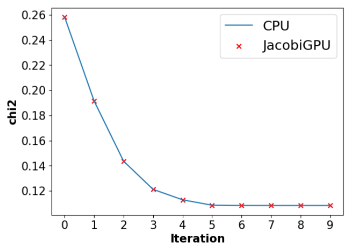
(d) Room3



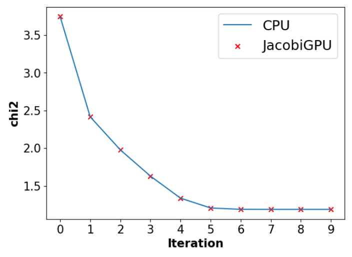
(e) Room4



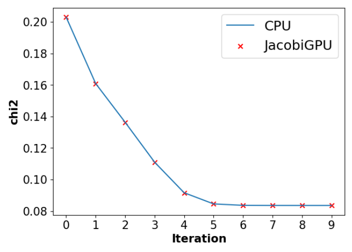
(f) Corridor1



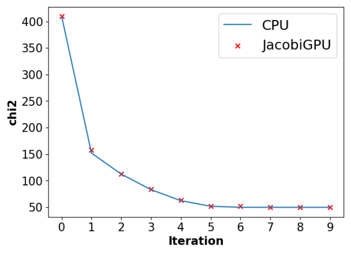
(g) Corridor5



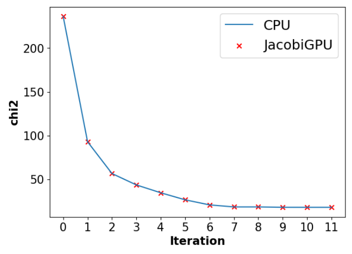
(h) Magistrale1



(i) Magistrale2



(j) Outdoors5



(k) Outdoors7

Figure 5.5: χ^2 error when we run the original CPU version and JacobiGPU integrated version of g2o block solver on outdoors7 sequence for ten iterations. Some taken from [25].

Chapter 6

Conclusion

In this thesis, we present a novel technique aimed at overcoming the computational challenges associated with linearization. This is crucial when numerical differentiation is used, a key component in the loop closure process of visual-inertial SLAM systems. Our approach, named JacobiGPU, leverages GPU resources to enhance computational efficiency. Unlike traditional methods that often rely on a singular partitioning strategy for graph-based computations, JacobiGPU innovatively partitions the graph based on computational tasks, thereby optimizing resource utilization and processing speed. This methodology has been integrated into ORB-SLAM3 with g2o, where the linearization process is offloaded to the GPU, representing a significant departure from conventional CPU-based processing.

The efficacy of JacobiGPU is evaluated using datasets from EuRoC and TUM-VI, encompassing both short and long sequences. The results demonstrate a notable average speedup of 2x in the loop closure process, with the linearization step showing an impressive improvement of up to 4x, particularly for longer sequences. These findings show the potential of leveraging GPU resources for enhancing the performance of visual-inertial SLAM systems, especially in the computationally intensive task of numerical differentiation.

Future Work

A potential direction for future research could involve conducting a comparative analysis between JacobiGPU and libraries that use automatic differentiation (auto-diff) techniques, with a focus on evaluating processing speed and memory consumption. Such a comparison would be instrumental in determining the most efficient method for linearization within SLAM systems, taking into account both computational efficiency and resource utilization.

Additionally, an interesting avenue for further exploration is the extension of JacobiGPU to optimize the solver component, especially within the framework of the Levenberg-Marquardt algorithm. As highlighted in Figure 5.1, the solver phase is the subsequent computationally demanding step following linearization. By focusing on optimizing this component, significant improvements in performance could be achieved. Investigating GPU-based approaches for enhancing solver efficiency has the potential to not only improve the overall effectiveness of SLAM systems but also establish new standards in processing speed and precision for real-time operations.

Bibliography

[1] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 3 2022.

[2] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE robotics & automation magazine*, 13(3):108–117, 2006.

[3] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016.

[4] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[5] Luca Carlone, Rosario Aragues, José A Castellanos, and Basilio Bona. A fast and accurate approximation for planar pose graph optimization. *The International Journal of Robotics Research*, 33(7):965–987, 2014.

[6] Luca Carlone, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. Initialization techniques for 3d slam: A survey on rotation estimation and its use in pose graph optimization. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4597–4604. IEEE, 2015.

[7] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[8] Jian S Dai. Euler–rodrigues formula variations, quaternion conjugation and intrinsic connections. *Mechanism and Machine Theory*, 92:144–152, 2015.

[9] Frank Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012.

[10] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[11] Taosha Fan and Todd Murphey. Generalized proximal methods for pose graph optimization. In *The International Symposium of Robotics Research*, pages 393–409. Springer, 2019.

[12] Jean Gallier and Dianna Xu. Computing exponentials of skew-symmetric matrices and logarithms of orthogonal matrices. *International Journal of Robotics and Automation*, 18(1):10–20, 2003.

[13] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. Openvins: A research platform for visual-inertial estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4666–4672. IEEE, 2020.

[14] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[15] Giorgio Grisetti, Rainer Kümmerle, Hauke Strasdat, and Kurt Konolige. g2o: A general framework for (hyper) graph optimization. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 9–13, 2011.

[16] Hashim A Hashim. Special orthogonal group so (3), euler angles, angle-axis, rodriguez vector and unit-quaternion: Overview, mapping and challenges. *arXiv preprint arXiv:1909.06669*, 2019.

[17] Nathan Jacobson. *Lie algebras*. Number 10. Courier Corporation, 1979.

[18] Anđela Jurić, Filip Kendeš, Ivan Marković, and Ivan Petrović. A comparison of graph optimization approaches for pose estimation in slam. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1113–1118. IEEE, 2021.

[19] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. isam: Incremental smoothing and mapping. *IEEE Transactions on Robotics*, 24(6):1365–1378, 2008.

[20] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos sycl for opencl: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, pages 1–1, 2015.

[21] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[22] Anthony W Knapp and Anthony William Knapp. *Lie groups beyond an introduction*, volume 140. Springer, 1996.

[23] Nikolaos Kourtzanidis and Sajad Saeedi. Rl-pgo: Reinforcement learning-based planar pose-graph optimization. *arXiv preprint arXiv:2202.13221*, 2022.

[24] Raimer Kress. *Numerical analysis*, volume 181. Springer Science & Business Media, 1998.

[25] Dhruv Kumar, Shishir Gopinath, Karthik Dantu, and Steven Y. Ko. Jacobigpu: Gpu-accelerated numerical differentiation for loop closure in visual slam. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024. to be published.

[26] Manolis IA Lourakis and Antonis A Argyros. Sba: A software package for generic sparse bundle adjustment. *ACM Transactions on Mathematical Software (TOMS)*, 36(1):1–30, 2009.

[27] E Pennestrì and R Stefanelli. Linear algebra and numerical algorithms using dual numbers. *Multibody System Dynamics*, 18:323–344, 2007.

[28] Ananth Ranganathan. The levenberg-marquardt algorithm. *Tutoral on LM algorithm*, 11(1):101–110, 2004.

[29] Jie Ren, Wenteng Liang, Ran Yan, Luo Mai, X Liu, and Xiao Liu. Megba: A high-performance and distributed library for large-scale bundle adjustment. In *European Conference on Computer Vision (ECCV)*, volume 2, 2022.

[30] David M Rosen, Luca Carlone, Afonso S Bandeira, and John J Leonard. Se-sync: A certifiably correct algorithm for synchronization over the special euclidean group. *The International Journal of Robotics Research*, 38(2-3):95–125, 2019.

[31] Antoni Rosinol, Marcus Abate, Yun Chang, and Luca Carlone. Kimera: an open-source library for real-time metric-semantic localization and mapping. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1689–1696, 2020.

[32] Antoni Rosinol, Andrew Violette, Marcus Abate, Nathan Hughes, Yun Chang, Jingnan Shi, Arjun Gupta, and Luca Carlone. Kimera: From slam to spatial perception with 3d dynamic scene graphs. *The International Journal of Robotics Research*, 40(12-14):1510–1546, 2021.

[33] David Schubert, Thore Goll, Nikolaus Demmel, Vladyslav Usenko, Jörg Stückler, and Daniel Cremers. The tum vi benchmark for evaluating visual-inertial odometry. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1680–1687. IEEE, 2018.

[34] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *Vision Algorithms: Theory and Practice: International Workshop on Vision Algorithms Corfu, Greece, September 21–22, 1999 Proceedings*, pages 298–372. Springer, 2000.

[35] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE, 2008.