2024 IEEE International Conference on Robotics and Automation (ICRA 2024)
May 13-17, 2024. Yokohama, Japan

JacobiGPU: GPU-Accelerated Numerical Differentiation for Loop Closure in Visual SLAM

Dhruv Kumar[1], Shishir Gopinath[1], Karthik Dantu[2], Steven Y. Ko[1]

***Abstract*— In this paper, we introduce JacobiGPU, a technique that uses a GPU to improve the efficiency of loop closure in visual-inertial SLAM systems, particularly when approximating Jacobians using the Finite Difference Method (FDM). Traditional FDM techniques often face computational overhead due to repeated perturbations in pose graphs. We address this overhead with a novel methodology, leveraging strategic graph partitioning and an optimized approach to Jacobian approximation. By integrating JacobiGPU into ORB-SLAM3's g2o, we enhance the linearization process. Our evaluation, conducted on 12 sequences of varying lengths from the EuRoC and TUM-VI datasets, demonstrated a speedup of up to 4.23x in the linearization stage and an overall enhancement of up to 2.08x in the overall optimization process.**

I. Introduction

Many modern robot perception tasks require Simultaneous Localization and Mapping [1], [2] to accurately estimate the robot's location as well as to build a representation of the environment. Most modern SLAM systems use vision sensors including cameras, LiDARs, and RGB-D sensors such as ORB-SLAM3 [3], Kimera [4], [5], Open-VINS [6], and others. A typical SLAM system performs visual odometry followed by loop closure which corrects for any drift when revisiting a previously visited location. Cameras capture images at 30fps and most LiDARs provide scans at 10Hz. To keep up with the sensors, it is important that all processing be performed promptly. In Visual SLAM, loop closure is typically the most computationally intensive module, and needs to be optimized for correct operation in real-time applications on resource-constrained hardware.

In several such systems, the map is represented as a pose graph that has robot poses as vertices and relative odometry as edge constraints between the vertices. Loop closure is performed as a pose graph optimization (PGO) problem which reconciles the constraints when a loop is discovered by minimizing overall error. This is solved as a non-linear least squares minimization problem typically using an iterative algorithm like Levenberg-Marquardt (LM) [7]. One approach to transform this non-linear optimization into a more manageable linear approximation is through the Finite Difference Method (FDM) [8].

FDM approximates the derivative of a function by using differences of the function evaluated at discrete points. For PGO, small perturbations are made to pose dimensions by a factor of δ, in both + and - directions to assess their effect on the cumulative error. While insightful, FDM is computationally intensive, primarily due to repeated perturbations for constraints with identical poses. Libraries like g2o [9] speed up the computation by employing parallelization strategies using APIs such as OpenMP, dividing the graph based on edges. However, this does not entirely solve the fundamental problem of redundant pose perturbations. This is because each pose is associated with multiple edges within the pose graph and a single perturbation is applied to each edge, resulting in multiple perturbations in total. We discuss this problem in more detail in Section IV.

To address the problem, we propose JacobiGPU, a new approach to FDM which harnesses the GPU's capabilities to perturb each pose only once. Our approach first partitions the graph based on *vertices* and performs perturbation, avoiding multiple perturbations per vertex. We then subsequently perform edge-centric computations for error evaluation. In doing so, we architect our approach to efficiently utilize the GPU and its memory with careful considerations on GPU memory management, GPU task (called *kernel*) management, and data transfer between the CPU and the GPU.

We evaluate JacobiGPU by integrating it with the g2o library used by ORB-SLAM3, where we offload the linearization step to JacobiGPU. Our experiments with 12 sequences from EuRoC [10] and TUM-VI [11] datasets show that we achieve a significant performance enhancement, with up to 76.35% or 4.23x speed increase for the linearization step and 51.95% or 2.08x increase for the entire optimization process. Moreover, we compare χ^2 error of the pose graph optimization and the SLAM trajectory between standard g2o and g2o integrated with JacobiGPU to show that other aspects of the optimization and the overall SLAM pipeline remain consistent. Our code is available at: https://github.com/sfu-rsl/jacobiGPU

II. Background

A pose graph representation captures the spatial evolution of a robot over time. Typically, a pose represents a position and rotation in 3D space, allowing for optimization over 6-DoF.

- **Vertices** correspond to poses at different points in time, and each encapsulates the robot's 3D position vector and 3×3 rotation matrix.
- **Edges** describe changes in relative position and rotation between poses. An edge may represent a sequential odometric constraint, linking two consecutive poses together, or a loop closure constraint, connecting a current pose back to an earlier one detected with the help of place recognition algorithms [12].

[1]Simon Fraser University
[2]University at Buffalo

A. Pose Graph Optimization

After constructing a pose graph G, the optimization can be modelled as a minimization of

$$\sum_{\langle i,j\rangle \in G} e_{ij}^T \Omega_{ij} e_{ij} \tag{1}$$

where e_{ij} is the error vector of a constraint between pose i and pose j and Ω_{ij} is the information matrix. Each e_{ij} is computed as

$$\begin{bmatrix} E_R \\ E_t \end{bmatrix} = \begin{bmatrix} \log_{\mathrm{SO(3)}}(R_w^i R_j^w R_i^j) \\ -R_w^i(R_j^w t_w^j) + t_w^i - t_j^i \end{bmatrix}. \tag{2}$$

Among the popular techniques to tackle this challenge are the Gauss-Newton method, which approximates the objective function with a quadratic and solves it linearly, and Gradient Descent, which iteratively adjusts parameters to minimize the objective function [13]. The Levenberg-Marquardt Algorithm stands out by combining techniques from both Gauss-Newton and Gradient Descent, yielding a more robust solution [14].

B. Linearization

By expressing a non-linear function by its Taylor series and retaining only the first-order term, we can derive a linear approximation that can be used to predict the system's behavior near a specific point. Thus, for a vector-valued multivariate function $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the first-order Taylor series expansion around a point $\mathbf{p}$ is:

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{p}) + \mathbf{J}(\mathbf{p})(\mathbf{x} - \mathbf{p}) \tag{3}$$

where $\mathbf{J}(\mathbf{p})$ is the Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{p}$. In PGO, the Jacobian captures the effect of pose perturbations on the error of each constraint, allowing the algorithm to steer towards a local minimum.

C. Finite Difference Method

The Finite Difference Method (FDM) is a numerical technique to estimate derivatives for Jacobians by perturbing input values and observing the subsequent change in the function. Utilizing the previously introduced vector-valued multivariate function $\mathbf{F}$, FDM captures the local slope by examining its values at nearby points. [8]. The central difference approximation for the derivative with respect to k^{th} dimension of jth pose is:

$$\frac{\partial F}{\partial x_{[j,k]}} \approx \frac{F(x_1, \ldots, x_k + \delta, \ldots) - F(x_1, \ldots, x_k - \delta, \ldots)}{2\delta}. \tag{4}$$

For PGO, FDM is integral because deriving an analytical Jacobian is challenging. Using the central difference approximation (4), small adjustments (perturbations) are made to just one pose parameter at a time, keeping the others constant, to approximate the derivative. This method facilitates the construction of one column of the Jacobian matrix at a time.

III. Related Work

In Graph-based SLAM, GTSAM [15] and g2o [9] are key libraries for optimization tasks like PGO. GTSAM employs factor graphs and Bayesian networks, with its iSAM [16] feature enabling efficient real-time updates in large-scale SLAM projects. Conversely, g2o enhances its PGO capabilities by incorporating Ceres Solver's automatic differentiation, complementing its foundational FDM approach.

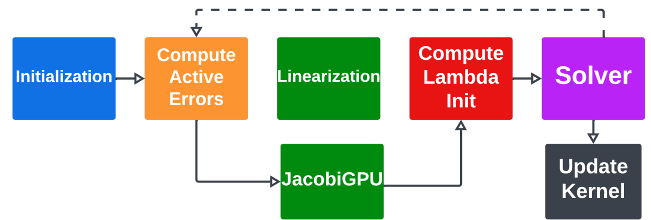

Fig. 1: Visual Representation of the Levenberg-Marquardt Algorithm: Highlighting JacobiGPU's Role in Offloading the Linearization Step.

A. Pose graph optimization

Fan et al. [17] pioneered a groundbreaking approach by generalizing proximal methods for special Euclidean groups, significantly accelerating the process of pose graph optimization. In their comparative studies, they demonstrated a remarkable 9x speedup over SE-Sync [18]. This is noteworthy, especially considering that Juri et al. [19] found SE-Sync to outperform g2o in simulated datasets [20]. However, the performance of SE-Sync on real-world visual inertial sequences remains an open question, making direct comparisons challenging.

In a different vein, Kourtzanidis et al. [21] ventured into the realm of deep reinforcement learning to optimize PGO. Their approach adeptly addresses the longstanding issue of algorithms becoming ensnared in local minima, particularly in 2D graphs. Parallel to this, other methodologies have emerged, such as those presented by Carlone et al. [22], which circumvent the need for initial estimates in optimization. These techniques demonstrate a marked speedup compared to the iterative solutions such as Levenberg-Marquardt algorithm. While their method surpasses the performance of g2o, its applicability is currently limited and does not extend to 3D visual inertial graphs.

B. Automatic Differentiation

Automatic Differentiation (AD) is a method that simplifies complex functions into basic operations for efficient computation of exact derivatives using the calculus chain rule. Agarwal et al. [21] employ AD in Ceres Solver to linearize constraints through residual blocks and cost functions. These are represented by Jets, a unique data structure in Ceres facilitating computation of residuals and derivatives, aiding iterative refinement. Ren et al. [22] expand this with SIMD-friendly JetVectors for GPU-based large-scale bundle adjustment. Nonetheless, AD has a significant memory overhead due to the need to retain intermediate variables for the backward pass. Dual numbers [23] further increase this, storing both function value and derivative [21]. Additionally, in situations like pose estimation, AD requires caution; proximate poses can create numerically unstable computations, posing challenges for optimization [23].

IV. Approach

During loop closure, the primary computational demands arise from the optimization phase (which we later show in Table II). Within this process, the linearization phase typically consumes the most time, particularly when Finite Difference Methods (FDM) is used to approximate Jacobians (which is also shown later in Figure 3). In this phase, the vertices associated with each edge undergo a slight perturbation by a factor of δ, resulting in a new vector. Subsequently, the derivative of the error is computed using this perturbed pose vector. Recognizing this bottleneck, our work focuses on offloading the linearization step to the GPU, as depicted in Figure 1. To improve efficiency in such scenarios, parallelization proves invaluable. A prime example is the g2o library [9], which leverages APIs like OpenMP to distribute subgraphs to individual threads via edge-based partitioning. However, even with such advancements, there are inherent limitations to this approach, as we explore next.

While parallelization strategies like those in the g2o library show promise, they do not fully address a fundamental inefficiency: the recurrent perturbation of identical poses. JacobiGPU addresses this by restructuring the FDM pipeline using specialized GPU kernels [24], ensuring each pose is perturbed only once per dimension. This recurring perturbation arises from the intrinsic structure of a pose graph, characterized by having more edges than unique poses, i.e., each pose is associated with multiple edges in the pose graph. This is due to multiple factors like overlapping observational regions, loop closures, and odometry-induced constraints. This results in poses being perturbed multiple times by their associated edges causing repeated computations.

A. Overview

JacobiGPU uses two specialized kernels — *perturb* (Section IV-C) and *compute* (Section IV-D) — each specifically designed for targeted tasks during linearization, addressing the problem of redundant perturbations. The perturb kernel partitions the graph based on vertices updating poses in both + and - directions simultaneously and storing the results in the green intermediate buffers, as depicted in Figure 2. This method ensures that every pose is adjusted only once, enhancing efficiency. Once this kernel completes its task, the compute kernel is launched, employing edge-based graph partitioning. This allows parallel error evaluations, ensuring the updated pose estimates align with the imposed constraints. From these, the Jacobian columns are derived and finally stored in the red buffer, also highlighted in Figure 2. This data is then ready for the CPU to use for solving. We also attempted to adapt the implementation by Gopinath et al. [25] to this PGO solver. However, for TUM-VI outdoors sequences, we found that the Hessian matrices generated by PGO were neither large nor dense enough to benefit from GPU-accelerated linear solvers.

The design principles of our work are rooted in best practices for GPU programming [24]. This involves careful consideration of several aspects: memory access patterns, data organization, task management, and data transfers. Unlike CPUs, which manage varied tasks and sporadic memory accesses, GPUs thrive on consistent memory access. To this end, JacobiGPU employs a straightforward data layout using one-dimensional buffers, avoiding complex multi-dimensional arrays that can disrupt uniform access. By organizing similar data types together, we optimize cache utilization through spatial locality. Moreover, while kernels are central to GPU computations, excessive launches can be counterproductive. Our FDM pipeline is designed to minimize kernel launches to reduce overheads. Furthermore, frequent data transfers between the CPU and GPU can be both vital and time-consuming. To mitigate this, JacobiGPU has a third kernel which updates poses directly on the GPU, minimizing the need for repetitive transfers and enhancing operational efficiency (Section IV-E).

B. Data Organization and Memory Architecture

GPUs prioritize efficient memory access and utilize a hierarchical memory structure with varying access speeds. One effective strategy to optimize kernel performance is by flattening multi-dimensional arrays, such as rotational matrices and translation vectors, into one-dimensional buffers. This ensures a more sequential data access pattern, which capitalizes on the GPU's memory bandwidth for faster execution. Building on this approach, we structured JacobiGPU with 18 distinct 1-D buffers each capturing various aspects of the pose graph and the multi-sensor system. The following is an overview of their functions.

Constraint and pose information: Constraint and pose information are stored across ten 1-D buffers. Two buffers store the rotational and translational data of the edges, capturing the relative poses between frames or nodes. The remaining eight buffers store Pose Data: four for the rotational matrices detailing orientation, and four for translation vectors. This data provides information about the position of the primary body in relation to the world and any linked cameras, as well as the relationship in terms of orientation and position between the body and the cameras. Storing these data types in distinct buffers simplifies uploads and optimizes cache use by leveraging spatial locality, making data access by multiple threads more efficient.

Intermediate perturbed pose: During the perturbation phase of pose estimation, adjustments in both + and - directions are needed for each dimension. Using only two intermediate buffers would mean perturbing in one direction, storing results with one kernel, and then computing the error with another. This process would be duplicated for the other direction, leading to 16 kernel launches for a 4DoF system, or 24 for 6DoF. However, JacobiGPU's optimized approach uses 4 intermediate buffers, capturing both perturbation results in a single launch. This reduces kernel launches to 8 or 12, albeit with a slight increase in memory use.

Jacobians: Two buffers, denoted by the red buffers in Figure 2, are used to store the Jacobians of the poses associated with each edge. One buffer is dedicated to the Jacobians of the source poses, while the other caters to the target poses. Organized for efficiency, these buffers can be indexed using edge IDs, allowing for swift data retrieval. Notably, the error kernel writes directly to these buffers,

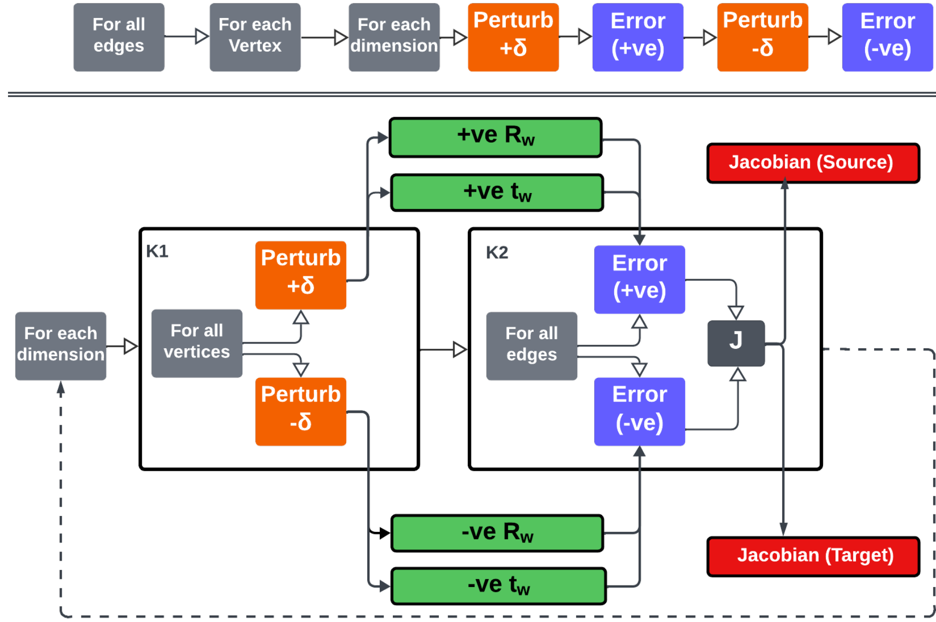

Fig. 2: g2o (top) vs JacobiGPU (bottom) linearization overview. Box labeled 'J' symbolizes the computation of the Jacobian

ensuring prompt and efficient updates of the corresponding Jacobians.

Pose IDs: In SLAM systems, keyframes, corresponding to poses, often do not have sequential IDs due to dynamic updates and inclusions. Non-sequential IDs disrupt GPU's optimized sequential access patterns. This unpredictability hinders effective prefetching, leading to cache misses and slower data retrieval in GPUs. JacobiGPU addresses this by methodically reindexing the keyframes upon integration, starting with an ID of zero. These reordered IDs are stored in two buffers, ensuring a consistent and organized sequence. These buffers can be indexed based on edge-ID allowing for immediate retrieval of both poses associated with each constraint. This arrangement simplifies understanding and navigating the relationships between poses within the specified constraints.

C. Perturb Kernel (K1)

The Perturb Kernel, depicted as K1 in Figure 2, is responsible for perturbing a pose in both + and - directions for a specific dimension. The resulting poses are then stored into their respective intermediate buffers, ready for subsequent optimization stages. Integrating this operation into a singular kernel launch presents several advantages: it maximizes GPU occupancy, leading to more uniform memory access and improved bandwidth use; it diminishes the overheads typically associated with multiple kernel launches, fostering a smoother operational flow. By streamlining these processes into one unified operation, the system leads to enhanced computational efficiency.

For computation, the kernel starts by considering the original world-to-body transformation and its associated translation. Perturbations $\delta\mathbf{R}$ and $\delta\mathbf{t}$ are introduced to capture changes in pose orientation and position, respectively. Subsequently, to transition from a world-centric to a body-centric perspective, this transformation is inverted. Finally, the relationship between the camera and the world is expressed via the camera-to-body transformation matrices, resulting in $\mathbf{R}_w$ and $\mathbf{t}_w$. Utilizing these transformations, the Jacobian column function can be reformulated as per Equation 4:

$$\frac{\partial \mathcal{L}}{\partial x_{j,k}} \approx \frac{\mathcal{L}(R^{j,k}_{w_{\text{pos}}}, t^{j,k}_{w_{\text{pos}}}) - \mathcal{L}(R^{j,k}_{w_{\text{neg}}}, t^{j,k}_{w_{\text{neg}}})}{2\delta} \quad (5)$$

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 kth 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 [26] is used to facilitate the transformation between matrices and vector spaces, and vice versa.

D. Compute Kernel (k2)

After perturbation, we proceed to determine the error between the predicted and actual poses based on Equation 2 for both + and - directions. Using the difference in these errors, divided by 2δ, we estimate the Jacobian as shown in Equation 5. Each dimension corresponds to a column in the Jacobian matrix. The kernel then locates the vertices connected to the edge, positioning each pose's column in the corresponding spot within the Jacobian buffer.

E. Update Kernel (k3)

The dedicated kernel operates on the GPU to seamlessly update all the poses within the optimization graph using spatial parameters from the solver as depicted in Figure 1. Critically, this updating process is conducted directly on the GPU side, eliminating the need to transfer new data between iterations,

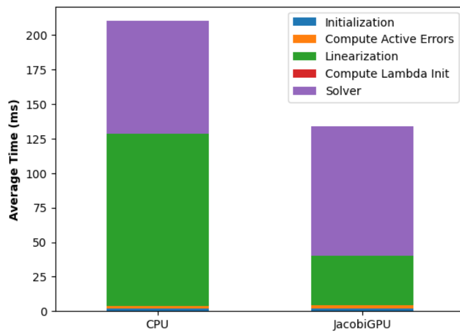

Fig. 3: A breakdown of a Levenberg-Marquardt call for loop closure in Outdoors5.

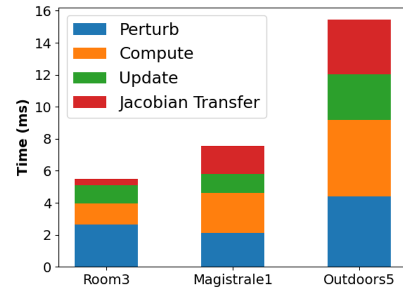

Fig. 4: Breakdown of various components within JacobiGPU for three different TUM-VI sequences.

thereby ensuring that the poses are instantly prepared for any subsequent steps in the optimization process.

The kernel's capability to directly update the poses on the GPU plays a pivotal role in drastically reducing data transfers. In iterative algorithms, where multiple rounds of updates are customary, transferring pose data after each iteration can be both time-consuming and resource-intensive. This kernel sidesteps that process entirely. By handling pose updates right where they are stored on the GPU, the continual need to shift large volumes of pose data across the system in every iteration is avoided. This not only speeds up the optimization process but also significantly reduces potential bottlenecks associated with iterative data movement.

V. Evaluation

A. Experimental Setup

Our experimental system uses a 12th Gen Intel(R) Core(TM) i7-12700K CPU with 20 cores at the maximum frequency of 5.0 GHz, 62GB of RAM, and NVIDIA GeForce RTX 3090 GPU. Our implementation is a modified version of g2o being used by ORB-SLAM3. We enable both vectorization and OpenMP. For evaluation, we run ORB-SLAM3 with sequences from EuRoC and TUM-VI that provide a loop closure of varying lengths with the stereo-inertial configuration. Further attesting to the validity of our approach, we compare the χ^2 error at each iteration and the trajectories between the baseline g2o and its modified version.

B. Run Time Performance

The integration of JacobiGPU demonstrates pronounced performance improvements, especially as the size of the graph grows. As shown in Table I, we observe up to 4.23x speedup in Jacobian calculations and up to 2.08x boost in the overall optimization process. It is noteworthy that our approach performs slightly worse for V103, primarily due to the graph size being small. In such instances, the initial data transfer cost can sometimes overshadow the computational benefits. As the graph size becomes bigger, we can observe larger and larger performance improvements.

As shown in Table II, the optimization within loop closure emerges as the bottleneck, which employs the Levenberg–Marquardt algorithm. Figure 3 presents an in-depth analysis of five crucial components of the algorithm, highlighting a notable speedup in linearization, which was our

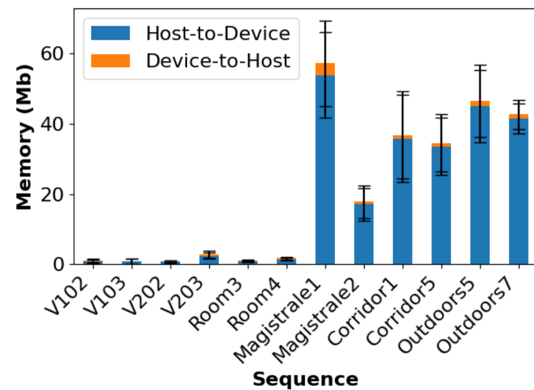

Fig. 5: Average memory transfer for buffer allocation across sequences, as profiled by Nsight Systems over five runs. The 'Magistrale1' bar aggregates the memory transfers for both loop closures 1 and 2.

focus, while the other components exhibit consistent performance. The solver time shows a slight increase, as this metric was obtained from two separate runs. Specifically for linearization, Figure 4 provides a detailed breakdown of the kernels within JacobiGPU, along with the cumulative data transfer time of the Jacobians to the CPU over all iterations. It is worth noting that the Jacobian transfer time for Magistrale1 is just under 2ms, while for outdoors5, it is around 4ms. Even though the graph size for outdoors5 — based on vertices and edges — is nearly three times larger than Magistrale1, the transfer time does not triple. This emphasizes the GPU's parallel processing capabilities and demonstrates that the transfer remains efficient, ensuring minimal overhead in the pipeline, as the graph size grows.

C. GPU Memory Transfer Analysis

We analyze memory transfers using NVIDIA Nsight Systems, profiling across various sequences (Figure 5). Among them, the 'Magistrale1' sequence demands the most significant GPU memory transfer, peaking at approximately 53.84MB. A significant portion of transfer originates from host-to-device relocation, as the entire graph's data is copied to the GPU for computation. Notably, the device-to-host transfer specifically for the Jacobian buffers is 3.4MB.

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 I: Linearization and total optimization run times (in ms) for ORB-SLAM3. The Magistrale1 sequence features two distinct loop closures, represented separately as Magistrale1(1) for the early stage and Magistrale1(2) for the later stage.

Components	CPU time (ms)	JacobiGPU time (ms)
Graph Construction	0.193	0.192
Optimization	210.159	137.185
Pose Recovery	0.041	0.042

TABLE II: Core components of loop closure recorded from one run outdoors7.

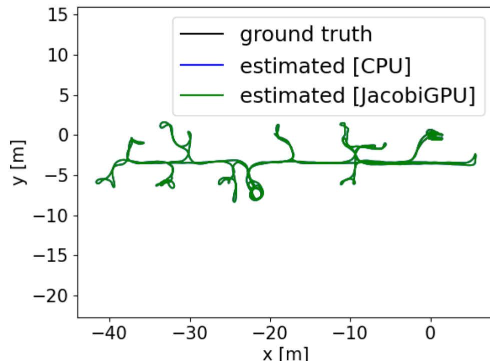

Fig. 6: ORB-SLAM3 trajectories from two distinct runs of corridor1: the original CPU-based implementation (blue) versus the enhanced version with JacobiGPU (green).

D. Trajectory

To analyze the trajectory data, we run the same sequence twice: first with the original ORB-SLAM3 configuration and then with our modified version. We run this experiment with all our sequences and Figure 6 shows one example. When examining the trajectories in both scenarios, they appear strikingly similar. However, we observe minor discrepancies in certain segments of the trajectory. It is important to note that these slight variations are not necessarily indicative of the performance of either implementation. Instead, they arise from the inherent variability that can occur across multiple runs of the same sequence. This variability is a common characteristic in SLAM systems and should be taken into account when comparing trajectories.

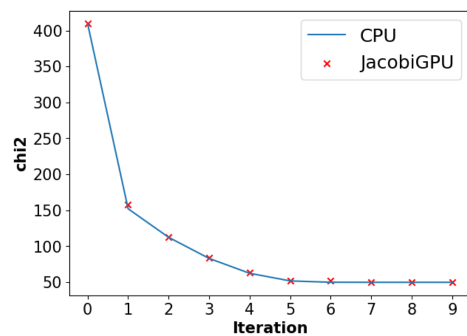

Fig. 7: χ^2 error when we run the original CPU version and JacobiGPU integrated version of g2o block solver on outdoors7 sequence for ten iterations.

E. Error Convergence

The χ^2 error from the g2o optimization, serves as a critical metric for assessing optimization accuracy and reliability. When we juxtapose the results from both the original ORB-SLAM3 and our version, a remarkable consistency emerges. As shown in Figure 7, the χ^2 error profiles for both implementations align perfectly. This indicates JacobiGPU reproduces the same behaviour as the original implementation.

VI. Conclusion

In this paper, we have discussed a technique to address the computational bottleneck inherent in numerical differentiation by utilizing GPU resources to optimize the loop closure procedure in visual-inertial SLAM. Our proposed solution, JacobiGPU, partitions the graph based on computational tasks, rather than adhering to a singular partitioning approach. We integrate JacobiGPU with g2o in ORB-SLAM3, offloading the linearization process to GPU. In our evaluation of both short and long sequences from the EuRoC and TUM-VI datasets, we observed an average 1.5x speedup in the loop closure process. Furthermore, the linearization step demonstrated up to a 4x improvement for the longer sequences.

References

[1] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics Automation Magazine*, vol. 13, pp. 99–110, June 2006.

[2] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (slam): part ii," *IEEE Robotics Automation Magazine*, vol. 13, pp. 108–117, Sep. 2006.

[3] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[4] A. Rosinol, A. Violette, M. Abate, N. Hughes, Y. Chang, J. Shi, A. Gupta, and L. Carlone, "Kimera: From slam to spatial perception with 3d dynamic scene graphs," *The International Journal of Robotics Research*, vol. 40, no. 12-14, pp. 1510–1546, 2021.

[5] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: an open-source library for real-time metric-semantic localization and mapping," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1689–1696, 2020.

[6] P. Geneva, K. Eckenhoff, W. Lee, Y. Yang, and G. Huang, "Openvins: A research platform for visual-inertial estimation," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4666–4672, 2020.

[7] A. Ranganathan, "The levenberg-marquardt algorithm," *Tutoral on LM algorithm*, vol. 11, no. 1, pp. 101–110, 2004.

[8] R. Kress, *Numerical analysis*, vol. 181. Springer Science & Business Media, 1998.

[9] G. Grisetti, R. Kümmerle, H. Strasdat, and K. Konolige, "g2o: A general framework for (hyper) graph optimization," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9–13, 2011.

[10] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016.

[11] D. Schubert, T. Goll, N. Demmel, V. Usenko, J. Stückler, and D. Cremers, "The tum vi benchmark for evaluating visual-inertial odometry," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1680–1687, IEEE, 2018.

[12] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, vol. 34, pp. 1004–1020, 8 2018.

[13] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—a modern synthesis," in *International workshop on vision algorithms*, pp. 298–372, Springer, 1999.

[14] "Sba: A software package for generic sparse bundle adjustment," *ACM Transactions on Mathematical Software*, vol. 36, 3 2009.

[15] F. Dellaert, "Factor graphs and gtsam: A hands-on introduction," *Georgia Institute of Technology, Tech. Rep*, vol. 2, p. 4, 2012.

[16] M. Kaess, A. Ranganathan, and F. Dellaert, "isam: Incremental smoothing and mapping," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1365–1378, 2008.

[17] T. Fan and T. Murphey, "Generalized proximal methods for pose graph optimization," in *The International Symposium of Robotics Research*, pp. 393–409, Springer, 2019.

[18] D. M. Rosen, L. Carlone, A. S. Bandeira, and J. J. Leonard, "Se-sync: A certifiably correct algorithm for synchronization over the special euclidean group," *The International Journal of Robotics Research*, vol. 38, no. 2-3, pp. 95–125, 2019.

[19] A. Jurić, F. Kendeš, I. Marković, and I. Petrović, "A comparison of graph optimization approaches for pose estimation in slam," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pp. 1113–1118, IEEE, 2021.

[20] L. Carlone, R. Tron, K. Daniilidis, and F. 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)*, pp. 4597–4604, IEEE, 2015.

[21] N. Kourtzanidis and S. Saeedi, "Rl-pgo: Reinforcement learning-based planar pose-graph optimization," *arXiv preprint arXiv:2202.13221*, 2022.

[22] L. Carlone, R. Aragues, J. A. Castellanos, and B. Bona, "A fast and accurate approximation for planar pose graph optimization," *The International Journal of Robotics Research*, vol. 33, no. 7, pp. 965–987, 2014.

[23] S. Agarwal, K. Mierle, and T. C. S. Team, "Ceres Solver," 3 2022.

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

[25] S. Gopinath, K. Dantu, and S. Y. Ko, "Improving the performance of local bundle adjustment for visual-inertial slam with efficient use of gpu resources," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6239–6245, 2023.

[26] N. Jacobson, *Lie algebras*. No. 10, Courier Corporation, 1979.