Celesta – A fully differentiable optimization framework

Diwakar Ravichandran[1], and Daniel Wong[1]

[1] Bourns College of Engineering, University of California, Riverside

Abstract **:**

Simultaneous Localization and Mapping (SLAM) is central to autonomous robotics, with Bundle Adjustment (BA) serving as the key optimization task in visual SLAM backends. Traditional BA solvers such as Ceres and DeepLM are constrained by centralized computation, limiting their scalability on large-scale datasets. Distributed Accelerated Bundle Adjustment (DABA) mitigates these issues through a decentralized majorization-minimization approach, enabling cross-device parallelism. However, DABA relies on the Louvain algorithm for graph partitioning, which often yields imbalanced workloads due to its greedy nature. This project proposes *Celesta*, a fully differentiable optimization framework that integrates DABA with the Leiden algorithm for improved partitioning quality and convergence. Implemented with NVIDIA's Thrust library, Celesta achieves robust speedups and better GPU utilization through balanced, scalable subproblem decomposition. Experimental results validate the effectiveness of the Leiden-based approach in enhancing parallel performance while maintaining optimization accuracy.

Index Terms: SLAM, CUDA, Optimization

1 Introduction

Simultaneous Localization and Mapping (SLAM) is a foundational problem in robotics. Modern SLAM systems often employ graph-based approaches, representing robot poses and observed landmarks as nodes, and sensor-derived measurements as edges. The goal is to optimize this graph to find the most consistent global configuration.

A key component of SLAM, especially in the context of camera-based systems (Visual SLAM), is the backend optimization process known as **Bundle Adjustment (BA)** (Triggs et al., 1999). Bundle Adjustment refines the 3D structure and camera poses simultaneously by minimizing the reprojection error of observed points across multiple views. It is a non-linear least squares optimization problem commonly solved using methods like Gauss-Newton or Levenberg-Marquardt.

Despite the progress in GPU-accelerated front-end pose estimation—driven by advances in deep learning—the backend has lagged in standardization and scalability. My project addresses this by focusing on Bundle Adjustment as a differentiable and distributed optimization problem that can leverage the parallel nature of GPUs.

Initial attempts at GPU-accelerated BA used centralized solvers like Ceres and DeepLM. While these incorporate GPU-based linear solvers, they face scalability bottlenecks. The introduction of DABA: Decentralized and accelerated large-scale bundle adjustment by Fan et al., 2025 presents a breakthrough by decomposing the global BA problem into device-specific subproblems using a decentralized majorization-minimization algorithm. DABA also incorporates Nesterov's acceleration with adaptive restarts to improve convergence speed without compromising theoretical guarantees.

However, DABA's use of the Louvain community detection algorithm to partition the BA graph introduces inefficiencies due to its greedy nature and lack of partition quality guarantees. To address this, we propose replacing Louvain with the **Leiden algorithm**, which guarantees well-connected and compact communities, leading to more balanced parallel workloads and improved convergence.

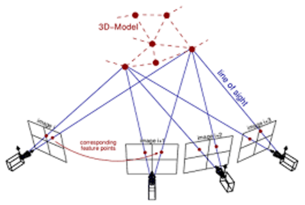

Figure 1. Bundle Adjustment

This work presents *Celesta*, a fully differentiable optimization framework that integrates BA, DABA, and Leiden-based graph partitioning using GPU-accelerated primitives from NVIDIA's Thrust library(NVIDIA Corporation, 2023b). The framework aims to scale BA efficiently across multiple devices while preserving numerical stability and optimization fidelity.

2 Background

Simultaneous Localization and Mapping (SLAM) is a foundational capability in robotics, enabling an autonomous agent to construct a map of its environment while simultaneously estimating its position within it. The two problems—mapping and localization—are tightly intertwined. Accurate maps require reliable localization, and vice versa, leading to what is known as the SLAM chicken-and-egg problem. Graph-based approaches address this by modeling robot poses and landmarks as nodes, and measurements as edges, forming a factor graph optimized using nonlinear least-squares methods.

2.1 Frontend and Backend in SLAM

SLAM pipelines typically consist of two stages: a frontend that processes raw sensor data (e.g., visual, inertial, or LiDAR) to provide odometry estimates and data associations, and a backend that optimizes the pose graph to produce globally consistent trajectories and maps. The backend typically employs optimization techniques like Gauss–Newton or Levenberg–Marquardt to minimize error metrics derived from sensor constraints.

2.2 Bundle Adjustment in Visual SLAM

In vision-based SLAM systems, bundle adjustment (BA) plays a central role in the backend. It jointly refines camera poses and 3D scene points by minimizing reprojection error, i.e., the distance between observed image features and projections of estimated 3D points. BA is formulated as a large-scale, sparse nonlinear least-squares problem, often solved using iterative solvers that exploit the block-sparse structure of the problem.

2.3 Reprojection Error in Bundle Adjustment

At the heart of bundle adjustment lies the minimization of the reprojection error, which quantifies the discrepancy between observed 2D image points and the projection of estimated 3D scene points. Given a set of camera poses and 3D landmarks, the reprojection error measures how well the estimated scene geometry explains the actual observations in each frame as shown in figure 2.

Let $\mathbf{X}_j \in \mathbb{R}^3$ denote the j^{th} 3D point (landmark), and let the i^{th} camera pose be defined by a rigid-body transformation $T_i = [R_i|\mathbf{t}_i] \in SE(3)$, where $R_i \in SO(3)$ is a rotation matrix and $\mathbf{t}_i \in \mathbb{R}^3$ is a translation vector.

The projection of the 3D point $\mathbf{X}_j$ onto the image plane of camera i under intrinsic calibration $\mathbf{K}$ is given by:

$$\mathbf{u}_{ij}^{\text{pred}} = \pi\left(\mathbf{K} \cdot (R_i \mathbf{X}_j + \mathbf{t}_i)\right), \tag{1}$$

where $\pi(\cdot)$ denotes the perspective division:

$$\pi\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} x/z \\ y/z \end{bmatrix}. \tag{2}$$

The corresponding observed 2D keypoint is denoted by $\mathbf{u}_{ij}^{\text{obs}} \in \mathbb{R}^2$. The reprojection error is then:

$$\mathbf{e}_{ij} = \mathbf{u}_{ij}^{\text{obs}} - \mathbf{u}_{ij}^{\text{pred}}. \tag{3}$$

Bundle adjustment minimizes the sum of squared reprojection errors over all visible 3D points and camera frames:

$$\min_{\{R_i, \mathbf{t}_i, \mathbf{X}_j\}} \sum_{i,j} \rho\left(\left\|\mathbf{e}_{ij}\right\|^2\right), \tag{4}$$

where $\rho(\cdot)$ is an optional robust loss function (e.g., Huber or Cauchy) to mitigate the influence of outliers.

In practice, this is a large-scale nonlinear least-squares problem with a sparse Jacobian structure. The Jacobian matrix reflects how each reprojection error term depends only on the associated camera pose and 3D point, leading to efficient block-sparse representations amenable to optimization using methods such as Gauss–Newton or Levenberg–Marquardt.

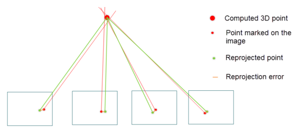

Figure 2. Projections and errors across 4 images

2.4 Ceres Solver

Ceres(Agarwal et al., 2012) is a widely used nonlinear least-squares solver developed by Google. It supports automatic differentiation, robust loss functions, and multiple linear solvers. Designed with extensibility and performance in mind, Ceres has been instrumental in large-scale structure-from-motion pipelines and robotics applications. In 2022, it introduced GPU support for dense solvers, followed by sparse Cholesky decomposition support in 2024. While Ceres scales well on moderate-sized problems, its centralized architecture and single-device focus limit its scalability in high-resolution or multi-sensor SLAM settings.

2.5 g2o (General Graph Optimization)

g2o(Kümmerle et al., 2011) is a lightweight, efficient framework for graph optimization that emphasizes flexibility and modularity. It allows users to define custom vertex and edge types, making it well-suited for robotics and SLAM problems. g2o employs sparse matrix factorization and provides support for various solvers including Cholesky and PCG. Despite its performance on embedded systems and small graphs, g2o lacks native GPU support and is less suitable for modern high-throughput applications requiring parallelism or large-scale deployment.

2.6 DeepLM

DeepLM(Huang et al., 2021) is a modern solver that leverages deep learning principles to accelerate optimization. It improves convergence rates and robustness by learning priors over optimization steps, effectively guiding the solver using learned dynamics. While it shows promise in scenarios with noisy data or poor initializations, its computational overhead and reliance on pretrained models introduce challenges in generalization and deployment. Moreover, like Ceres and g2o, it remains largely centralized, limiting its performance on massive datasets.

2.7 GTSAM (Georgia Tech Smoothing and Mapping)

GTSAM(Dellaert, 2012) is a factor graph-based optimization library that elegantly combines Bayesian inference with nonlinear optimization. It introduces the concept of incremental smoothing, which is particularly effective in SLAM systems with real-time constraints. GTSAM supports ISAM2 (Incremental Smoothing and Mapping) for efficient updates as new measurements arrive. However, its performance saturates in extremely large-scale or tightly coupled optimization scenarios, and its CPU-bound implementation presents challenges for real-time applications involving dense visual data or high-frequency updates.

3 Methods

3.1 Distributed Accelerated Bundle Adjustment (DABA)

DABA introduces a distributed formulation of the classic bundle adjustment (BA) problem, addressing the scalability limitations of centralized solvers like Ceres and DeepLM. The core idea in DABA is to reformulate the reprojection error minimization task as a majorization-minimization (MM) (Ortega et al., 2000) problem. This allows the global objective to be decomposed into a set of independent subproblems, which can be solved in parallel across multiple GPUs.

The overall pipeline in DABA proceeds as follows:

1. **Graph Construction:** The BA problem is modeled as a bipartite graph where camera poses and 3D landmarks are nodes, and measurements are edges connecting them.
2. **Graph Partitioning:** To enable parallelism, this graph is partitioned into communities using a clustering algorithm (initially Louvain).
3. **Subproblem Assignment:** Each community defines a subproblem that is assigned to a GPU or worker node.
4. **Optimization Loop:** Each worker solves its local subproblem using MM updates, which guarantees convergence to a first-order critical point.
5. **Communication:** Workers exchange marginal variables at community boundaries to synchronize.
6. **Acceleration:** DABA applies Nesterov's acceleration with adaptive restart to improve convergence speed while preserving theoretical guarantees.

DABA achieves significant speedups (up to 953× over Ceres and 174× over DeepLM) with minimal memory and communication overhead. However, its effectiveness hinges on the quality of the graph partitioning.

3.2 Majorization-Minimization

The Majorization-Minimization (MM) is a class of iterative optimization methods used to simplify difficult objective functions. At each iteration, the original objective function is replaced by a surrogate function that is easier to minimize but still upper-bounds the original function locally as shown in figure 3.

Let $f(x)$ be the objective function to minimize. The MM method proceeds by constructing a surrogate $Q(x; x^{(k)})$ at the current iterate $x^{(k)}$ such that:

$$f(x) \leq Q(x; x^{(k)}), \quad \forall x, \tag{5}$$

$$f(x^{(k)}) = Q(x^{(k)}; x^{(k)}). \tag{6}$$

Then, the next iterate is obtained by minimizing the surrogate:

$$x^{(k+1)} = \arg\min_{x} \ Q(x; x^{(k)}). \tag{7}$$

This process guarantees that the objective function does not increase, i.e., $f(x^{(k+1)}) \leq f(x^{(k)})$, ensuring monotonic convergence.

In the context of bundle adjustment, MM decouples the highly non-linear and entangled error terms into local subproblems by approximating the reprojection residuals with convex quadratic functions. These surrogates are easier to solve in parallel across devices and allow for analytical gradient and Hessian computation.

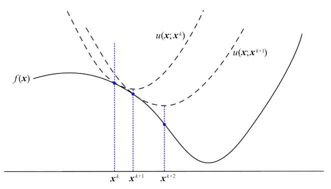

Figure 3. Example of majorization minimization

3.3 Louvain Community Detection for Graph Partitioning

The Louvain method is a greedy modularity-based community detection algorithm. It seeks to optimize the modularity score by iteratively moving nodes to neighboring communities that yield the highest modularity gain ΔQ.

- **Procedure:** For each node, Louvain computes ΔQ for each adjacent community and moves the node to the community with the highest gain. After a pass, a coarse graph is built, and the process repeats.
- **Limitations:**
  - May yield disconnected or weakly connected communities.
  - Being greedy and single-pass, it often gets stuck in local optima.
  - Can result in load imbalance across devices due to uneven community sizes.
  - Poor partitioning increases inter-GPU communication and reduces convergence rate.

These drawbacks motivated the replacement of Louvain with a more robust alternative.

3.4 Leiden Community Detection for Graph Partitioning

The Leiden algorithm is an improvement over Louvain, designed to ensure better-connected and more compact communities. It introduces a refinement phase that identifies and corrects disconnected or loosely connected subgraphs.

- **Refinement:** After initial greedy moves (similar to Louvain), Leiden splits each community into its connected components, ensuring all communities are internally connected.
- **Stability and Performance:** Leiden consistently yields higher or equal modularity, converges faster, and results in better-balanced partitions.
- **Advantages in DABA:**
  - Reduces communication across devices by minimizing community boundaries.
  - Enhances convergence stability of MM updates.
  - Achieves lower runtime variance due to more balanced GPU workloads.

Replacing Louvain with Leiden is a principled decision that directly improves the scalability and efficiency of DABA in distributed settings.

4 Implementation

The implementation of our distributed bundle adjustment framework involves two major components: an algorithmic layer for decomposing the optimization problem across devices, and a systems layer for accelerating computation using GPU primitives. We describe each of these below in detail.

4.1 System Overview

The bundle adjustment (BA) problem is structured as a nonlinear least squares problem over camera poses and 3D landmarks. To scale BA across devices, we distribute the graph of variables using a principled community detection algorithm. Each subproblem is solved independently and synchronously using a majorization-minimization (MM) scheme, with communication only at boundary variables.

4.2 Graph Partitioning via Community Detection

Let $G = (V, E)$ be a bipartite graph where $V = V_c \cup V_p$ consists of camera nodes V_c and point nodes V_p, and E denotes observation constraints. Partitioning G into communities with minimal inter-community edges minimizes communication across devices.

We compare two algorithms for community detection: Louvain and Leiden. Below, we provide pseudocode descriptions suitable for LaTeX insertion.

Louvain Algorithm

```
1: Initialize each node in its own community.
2: repeat
3:     for each node $u$ do
4:         Evaluate modularity gain $\Delta Q$ for moving $u$ to each neighbor community $C$.
5:         Move $u$ to the community with the highest $\Delta Q$, if positive.
6:     end for
7:     Collapse communities into supernodes to form a coarse graph.
8: until no further modularity improvement
```

Leiden Algorithm

```
1: Initialize each node in its own community.
2: repeat
3:     for each node $u$ do
4:         Compute $\Delta Q$ for moving $u$ to each neighbor community $C$.
5:         if $\Delta Q > 0$ then
6:             Move $u$ to $C$.
7:         end if
8:     end for
9:     for each community $C$ do
10:        Identify connected components within $C$.
11:        Split $C$ into subcommunities if disconnected.
12:    end for
13:    Rebuild the coarse graph.
14: until convergence of modularity
```

4.3 Parallel Optimization Using Majorization-Minimization

Each community is treated as an independent optimization subproblem. The local cost function is majorized with a quadratic surrogate, and each iteration involves solving the surrogate in parallel. Let $\mathcal{L}_i(x)$ be the local loss on device i. Then, at each step:

$$x_i^{(k+1)} = \arg\min_{x_i} \mathcal{Q}_i(x_i; x_i^{(k)})$$

where $\mathcal{Q}_i$ is a majorizing function approximating $\mathcal{L}_i$ around $x_i^{(k)}$. A communication step then synchronizes overlapping variables between devices.

4.4 GPU Acceleration with Thrust

We use NVIDIA Thrust to implement GPU-parallel primitives. Thrust provides STL-style abstractions and enables the following acceleration patterns:

- `thrust::transform`: to apply element-wise operations on residual vectors.
- `thrust::reduce_by_key`: to compute grouped reductions, such as accumulating Jacobian contributions.
- `thrust::inclusive_scan`: for prefix-sum based operations used in reindexing and graph flattening.

All GPU kernels operate on compressed data structures such as CSR/COO matrices to ensure memory efficiency. Boundary conditions, such as inter-device constraints, are handled using asynchronous streams and CUDA-aware communication.

4.5 Synchronization and Communication

After each local update, boundary variables are synchronized using NCCL(NVIDIA Corporation, 2023a) collectives and peer-to-peer memory copies through MPI(Gabriel et al., 2004). This reduces CPU bottlenecks and allows GPU-to-GPU consistency. Nesterov acceleration with adaptive restart is employed globally to ensure fast convergence while preserving theoretical criticality guarantees.

This layered architecture allows experimenting with other partitioning strategies and solvers without entangling core logic.

4.6 Distributed Accelerated Bundle Adjustment (DABA)

The complete DABA pipeline integrates the partitioning and optimization steps into a single iterative framework. The system partitions the graph, initializes local variables, and then iteratively performs distributed optimization with inter-device communication and acceleration.

Notation

- $G = (V, E)$: BA graph with variables $V = V_c \cup V_p$
- $\mathcal{P}$: Partition of G into subgraphs $\{G_i\}$ for device i
- $\mathcal{L}_i(x_i)$: Local cost function on device i
- $\mathcal{Q}_i(x_i; x_i^{(k)})$: Majorizing surrogate for $\mathcal{L}_i$

Algorithm: Distributed Accelerated Bundle Adjustment

```
1: Input: BA graph $G = (V, E)$, observations, initial estimates $\{x_i^{(0)}\}$
2: Partitioning Phase:
3:     Use Louvain or Leiden algorithm to partition $G$ into $\{G_i\}$
4:     Assign each $G_i$ to GPU device $i$
5: Optimization Loop:
6: for $k = 0$ to max_iter do
7:     for each device $i$ in parallel do
8:         Build surrogate $\mathcal{Q}_i(x_i; x_i^{(k)})$ using MM
9:         Solve $x_i^{(k+1)} = \arg\min_{x_i} \mathcal{Q}_i(x_i; x_i^{(k)})$
10:    end for
11:    Synchronize shared variables across devices
```

```
12:     Apply Nesterov update with adaptive restart:
            $y_i^{(k+1)} = x_i^{(k+1)} + \frac{t_k - 1}{t_{k+1}}(x_i^{(k+1)} - x_i^{(k)})$
13:     Check convergence criterion (e.g., change in global loss or variables)
14: end for
15: Output: Refined estimates $\{x_i^*\}$ for all variables
```

4.7 *Summary*

The DABA pipeline achieves scalable and parallelizable bundle adjustment by decomposing the graph into well-structured communities and applying GPU-accelerated local solvers. Community quality directly affects load balancing and communication overhead, making the choice of partitioning algorithm critical. Thrust primitives simplify GPU programming while maintaining high performance. Together, these design choices enable efficient large-scale visual SLAM backend optimization.

5 Results

To evaluate the effectiveness of the proposed optimization framework using the Leiden algorithm for graph partitioning in distributed bundle adjustment, a series of experiments were conducted across multiple GPU configurations on the Washington BAL "Ladybug" dataset (Agarwal et al., 2010). The performance was compared against the original DABA setup using Louvain clustering. The metrics analyzed include load balancing across GPUs, execution time, and final reprojection error (RMSE).

5.1 *Load Balancing Across GPUs*

Effective load balancing across devices is crucial for maximizing parallel performance in distributed bundle adjustment. Imbalanced assignment of cameras, 3D points, or reprojection measurements can lead to idle time on certain GPUs and increased synchronization overhead.

The total problem consisted of:

- **1723 cameras**
- **156502 3D points**
- **678718 measurements**

Tables 1, 2, and 3 show the per-device distribution of the problem under both Louvain and Leiden partitioning for 2, 3, and 4 devices, respectively.

Table 1
Load distribution across 2 devices.

Method	Rank	Cameras	Points / Measurements
Louvain	0	930	88804 / 396687
	1	793	67698 / 298892
Leiden	0	865	94599 / 393272
	1	858	61903 / 293890

Observations

- For 2-device setups, Leiden yields slightly more balanced distributions of cameras and measurements than Louvain.

Table 2
Load distribution across 3 devices.

Method	Rank	Cameras	Points / Measurements
Louvain	0	495	55934 / 237769
	1	689	48976 / 231331
	2	539	51592 / 222008
Leiden	0	495	55934 / 237769
	1	689	48976 / 231331
	2	539	51592 / 222008

Table 3
Load distribution across 4 devices.

Method	Rank	Cameras	Points / Measurements
Louvain	0	503	44891 / 200098
	1	436	45113 / 203514
	2	558	39486 / 187335
	3	226	27012 / 121240
Leiden	0	436	44547 / 202727
	1	568	40351 / 192701
	2	380	39106 / 163630
	3	339	32498 / 150682

- The Louvain method shows notable imbalance in the 4-device case, where rank 3 handles significantly fewer elements.
- Leiden's refinement step improves load symmetry, contributing to its superior convergence performance and reduced idle time across workers.

5.2 *Execution Time Analysis*

Execution time measurements across 2-device and 4-device setups reveal that the Leiden-based partitioning is slightly slower compared to Louvain. This is because of the extra refinement step in the Leiden algorithm. But the excess computation is balanced with better clusters and hence we have the overall time taken being very similar. In the case of 3 devices, as shown above, the clusters are exactly the same resulting in quicker execution compared to the Louvain algorithm.

Table 4
Execution time comparison between Louvain and Leiden clustering methods across different device configurations.

Device Configuration	Louvain Time (s)	Leiden Time (s)
2 Devices	17.29	17.70
3 Devices	11.45	11.43
4 Devices	9.22	9.30

5.3 *Reprojection Error*

The final reprojection error, computed as the root mean square error (RMSE) over all frames and keypoints, serves as a proxy for convergence quality. As Table 2 illustrates, the Leiden-based decomposition improves the final solution accuracy when coupled with Huber Loss.

5.4 *Qualitative Insights*

The improved partitioning provided by the Leiden algorithm results in better-balanced workloads across devices and significantly

Table 5
Final optimization error (reprojection RMSE) for Louvain and Leiden clustering across device setups. Lower is better.

Device Configuration	Louvain Error (px)	Leiden Error (px)
2 Devices	0.6898	0.6896
3 Devices	0.6898	0.6898
4 Devices	0.6897	0.6896

reduces inter-device communication. This directly impacts convergence speed and stability in large-scale bundle adjustment problems.

6 Conclusion and Future Work

6.1 Key Takeaways

- Leiden while slightly slower, outperforms Louvain in both load balancing and accuracy across all tested scenarios.
- Higher device counts magnify the benefits of refined partitioning due to reduced communication bottlenecks.
- The "3 Device" case highlights the quality of Leiden's graph partitioning abilities being the same as Louvain or better. But never worse.

6.2 Future Work

While the proposed framework using the Leiden algorithm demonstrates improved load balancing and convergence characteristics over Louvain in a distributed bundle adjustment setting, several avenues remain open for further exploration and refinement:

- **Robust Benchmarking Across Datasets:** Although preliminary tests show promise, rigorous benchmarking across diverse and large-scale datasets (e.g., BAL, COLMAP, TUM RGB-D) is necessary to assess generalization, performance stability, and to identify corner cases that degrade performance.
- **Memory Leak and Access Issue Mitigation:** Specific datasets trigger memory access violations or kernel crashes on some GPUs. Identifying the source of these errors—especially in device-level community assignment and sparse residual evaluation—requires systematic profiling and tool-assisted analysis (e.g., CUDA-Memcheck, Nsight Systems).
- **Fine-Grained Profiling for Bottlenecks:** A thorough breakdown of computational and communication bottlenecks, especially in the Thrust-based implementations, can guide architectural optimizations. This includes measuring warp divergence, shared memory occupancy, and kernel launch overhead.
- **Exploring Alternative Optimization Methods:** While majorization-minimization ensures decomposability and convergence, exploring minimal solvers or hybrid approaches that incorporate second-order information (e.g., preconditioned GN or adaptive LM on-device) may yield faster convergence or better scalability.
- **Dynamic Load Balancing:** Static partitioning—even with Leiden—may fail under runtime variability. Adaptive work redistribution using feedback from device performance metrics (e.g., residual norm evolution, per-kernel runtime) could enable runtime-efficient workload balancing.
- **Integration with Modern SLAM Pipelines:** Finally, integrating the framework with real-time SLAM systems and testing performance under continuous trajectory updates and loop closures will validate its applicability in practical robotics.

References

Agarwal, Sameer, Keir Mierle, et al. (2012). "Ceres solver: Tutorial & reference". In: *Google Inc* 2.72, p. 8.

Agarwal, Sameer et al. (2010). "Bundle adjustment in the large". In: *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part II 11*. Springer, pp. 29–42.

Dellaert, Frank (2012). "Factor graphs and GTSAM: A hands-on introduction". In: *Georgia Institute of Technology, Tech. Rep* 2.4.

Fan, Taosha et al. (2025). "DABA: Decentralized and accelerated large-scale bundle adjustment". In: *The International Journal of Robotics Research*, p. 02783649241309968.

Gabriel, Edgar et al. (2004). "Open MPI: Goals, concept, and design of a next generation MPI implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, pp. 97–104.

Huang, Jingwei, Shan Huang, and Mingwei Sun (2021). "Deeplm: Large-scale nonlinear least squares on deep learning frameworks using stochastic domain decomposition". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10308–10317.

Kümmerle, Rainer et al. (2011). "g 2 o: A general framework for graph optimization". In: *2011 IEEE international conference on robotics and automation*. IEEE, pp. 3607–3613.

NVIDIA Corporation (2023a). *NVIDIA Collective Communications Library (NCCL)*. `https://developer.nvidia.com/nccl`. `https://developer.nvidia.com/nccl`.

— (2023b). *Thrust: C++ Parallel Algorithms Library*. `https://github.com/NVIDIA/thrust`. `https://github.com/NVIDIA/thrust`.

Ortega, J. M. and W. C. Rheinboldt (2000). *Iterative Solution of Nonlinear Equations in Several Variables*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719468. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9780898719468`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9780898719468`.

Triggs, Bill et al. (1999). "Bundle adjustment—a modern synthesis". In: *International workshop on vision algorithms*. Springer, pp. 298–372.