Graphite: A GPU-Accelerated Mixed-Precision Graph Optimization Framework

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

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

ICRA IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION

Introduction

Graphite is a GPU-accelerated **nonlinear least squares optimization framework** for estimation processes in computer vision and robotics such as **simultaneous localization and mapping (SLAM)**.

- Keyframe-based visual-inertial SLAM refines **poses, map points, and inertial variables** by minimizing residuals over **constraints**.
- These optimization tasks often **run repeatedly** and must complete **as quickly as possible**.
- Modern robotics platforms are equipped with **GPUs** which excel at processing thousands of tasks in parallel.
- Existing GPU solvers are limited to simple numeric variables, rely on domain-specific languages, or target a fixed class of problems.
- However, SLAM systems require support for **complex user-defined optimizable variables** and constraints involving as many as **six** variables each [1].
- On **resource-constrained platforms**, GPU memory is often **limited** and **shared with the CPU**, making memory efficiency critical.

Graphite addresses these challenges via an explicit GPU-friendly batching design and efficient mixed-precision solving techniques.

Workflow

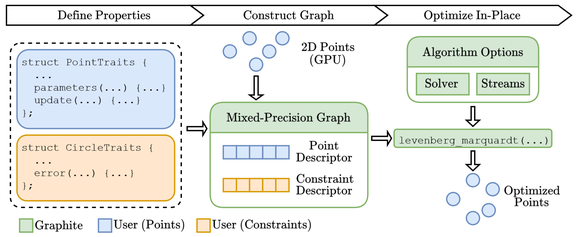


Figure 1. An example workflow for optimizing 2D points.

To optimize variables using Graphite, there are three main steps.

1. The user defines **static properties** for variables and constraints.
2. Variables and constraints are arranged into batches represented by **descriptors** which are used to build an **optimizable graph**.
3. The graph is iteratively optimized, which refines the variables.

Background

An optimization over constraints ($\mathcal{C}$) aims to find parameters (x) which minimize the total error

$$\arg\min_x \frac{1}{2} \sum_{\langle i,j \rangle \in \mathcal{C}} r_{ij}^\top \Sigma_{ij}^{-1} r_{ij}$$

where r_{ij} is the residual for a constraint between variables i and j and Σ_{ij} is its covariance matrix. **Levenberg-Marquardt** solves this iteratively. Each iteration finds a step Δx by solving a linear system

$$\underbrace{(J^\top \Sigma^{-1} J + \lambda D^\top D)}_{H_\lambda} \Delta x = -\underbrace{J^\top \Sigma^{-1} r}_{b}$$

where r is the stacked residual with covariance Σ, $J = \frac{\partial r}{\partial x}$ is the Jacobian, H_λ is the damped Hessian, and b is the gradient. The parameter λ is a damping factor and $D^\top D$ is $\text{diag}(J^\top \Sigma^{-1} J)$ or I.

Design Overview

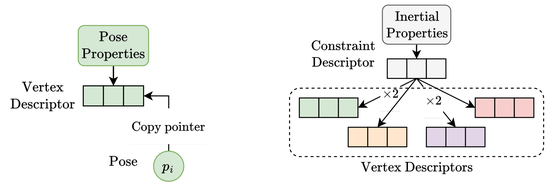
Graphite represents an optimization problem as a **graph of descriptors**, and solves it on the GPU using the Levenberg-Marquardt algorithm. It provides a CUDA C++ interface to support SLAM development.

- **Descriptor Batching** — variables and constraints of the same type are grouped into **descriptors** for efficient GPU processing.
- **Differentiation Modes** — Jacobians can be evaluated via user-supplied analytic expressions, dynamically (matrix-free), or via forward-mode automatic differentiation.
- **Mixed-Precision Solver** — the parameter step Δx is computed quickly and efficiently using mixed-precision preconditioned conjugate gradients.

Descriptor Batching

Variables (vertices) and constraints of the **same type** are grouped into **descriptors**, which enable GPU-friendly batching of uniform work.

- **Vertex descriptors** specify the parameter block size, parameterization and update functions, and the data type. They store pointers to variable data, enabling **in-place optimization**.
- **Constraint descriptors** specify error and Jacobian functions, residual size, observation and auxiliary variable data types, required vertex descriptors, the loss function, and the differentiation mode.
- These properties are used to **specialize CUDA kernels** at **compile time**, so every thread in a batch performs identical computations, reducing adverse performance effects from branching.



(a) A vertex descriptor stores pointers to variables.

(b) A constraint descriptor links one or more vertex descriptors.

Figure 2. Examples of vertex and constraint descriptors.

Differentiation Modes

Graphite supports three modes for computing **Jacobian matrices**.

- **Analytic** — Jacobians are evaluated from user-defined expressions concurrently across multiple CUDA streams.
- **Dynamic (matrix-free)** — Jacobians are evaluated from expressions on-the-fly at the point of use (e.g. matrix-vector products), reducing memory usage.
- **Automatic** — Jacobians are evaluated via forward-mode automatic differentiation. Each CUDA thread computes one Jacobian column by passing dual numbers to the residual function of a constraint.

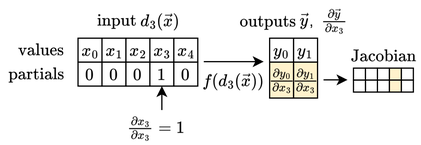


Figure 3. Forward-mode autodiff using dual numbers.

Mixed-Precision Solver

To solve Δx, Graphite applies **preconditioned conjugate gradients (PCG)**. Graphite allows the **vertex and constraint precision** and **linear system precision** to be configured independently, enabling trade-offs between convergence, runtime, and GPU memory usage.

- Supported precisions are **FP64, FP32, and BF16** (e.g. FP32 for vertices and constraints, and BF16 for the linear system).
- The PCG solver avoids the costs of building and storing the full Hessian matrix by evaluating products implicitly using the Jacobian.
- To maintain convergence at lower precisions, Graphite applies Hessian diagonal **clamping**, Jacobian column **rescaling**, and PCG residual **normalization** each iteration. The precision of the block Jacobi preconditioner is also upgraded when it is too low.
- The Jacobian matrix structure is inferred from the graph itself to avoid the overhead of matrix format conversion.

Evaluation Setup

We perform experiments on a desktop machine and an NVIDIA Jetson Orin Nano Super embedded developer kit.

Table 1. Evaluation machine specifications.

Machine	Specifications
Desktop	12-core Intel Core i7-12700K CPU @ 3.8-4.9 GHz 10496-core NVIDIA RTX 3090 GPU @ 1.9 GHz 64 GB RAM
Jetson Orin Nano Super	6-core ARM Cortex-A78AE CPU @ 1.7 GHz 1024-core NVIDIA Ampere GPU @ 1.0 GHz 8 GB RAM

Bundle Adjustment Performance

We evaluate Graphite on Bundle Adjustment in the Large (BAL) datasets [2] against several libraries [3, 4, 5] on the desktop machine.

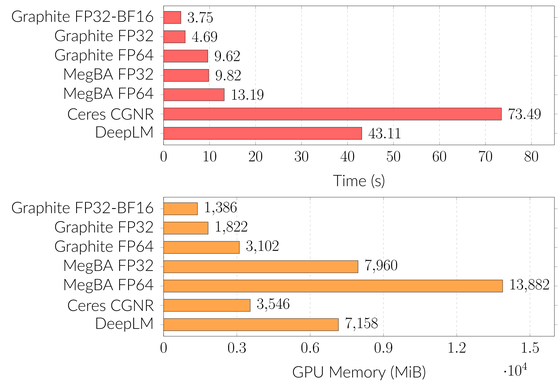


Figure 4. Final-4585: Comparison across solvers.

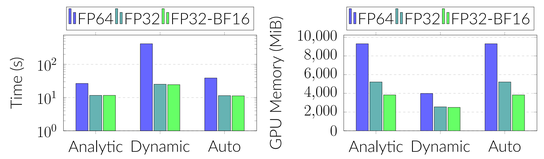


Figure 5. Final-13682: Comparison across differentiation modes.

Case Study: Visual-Inertial SLAM

We integrate Graphite into **ORB-SLAM3** [1] for **global visual-inertial bundle adjustment**, and evaluate it on large maps generated by outdoors **TUM-VI** dataset sequences [6].

- The optimization involves **5 variable types** (pose, velocity, gyro and acc. biases, and map points) and **7 constraint types** (mono/stereo reprojection errors, inertial and bias residuals, and bias priors).
- The graph uses lightweight **CUDA-compatible data structures** in place of the original classes for GPU optimization.

Table 2. Problem sizes. Note that 4 vertices are created per pose.

	Desktop			Orin Nano Super		
Sequence	Poses	Points	Constraints	Poses	Points	Constraints
outdoors1	3,131	58,886	348,548	2,483	48,621	278,661
outdoors2	2,032	57,641	338,364	1,972	33,831	213,125
outdoors3	1,595	36,536	272,307	1,589	24,524	174,070

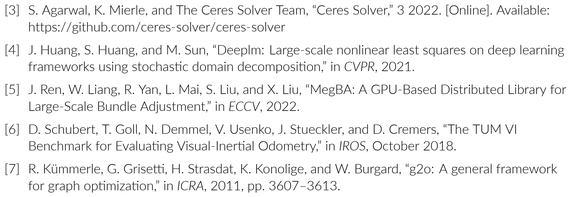


Figure 6. Relative speedup of Graphite (FP64) vs g^{2}o PCG [7] for global visual-inertial bundle adjustment on generated maps.

Summary

Graphite enables fast, memory-efficient, large-scale optimization over complex constraints for desktop and resource-constrained devices.

- Evaluated on **BAL** datasets, Graphite achieves similar performance to MegBA while using up to **78% less GPU memory**.
- Applied to **global visual-inertial bundle adjustment** in ORB-SLAM3, Graphite achieves speedups of up to $\sim 59\times$ over a CPU baseline on desktop, and up to $\sim 6\times$ on the Jetson Orin Nano Super.

`https://github.com/sfu-rsl/graphite`

References

[1] 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.

[2] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *ECCV*. Springer, 2010, pp. 29–42.

[3] S. Agarwal, K. Mierle, and The Ceres Solver Team, "Ceres Solver," 3 2022. [Online]. Available: https://github.com/ceres-solver/ceres-solver

[4] J. Huang, S. Huang, and M. Sun, "Deeplm: Large-scale nonlinear least squares on deep learning frameworks using stochastic domain decomposition," in *CVPR*, 2021.

[5] J. Ren, W. Liang, R. Yan, L. Mai, S. Liu, and X. Liu, "MegBA: A GPU-Based Distributed Library for Large-Scale Bundle Adjustment," in *ECCV*, 2022.

[6] D. Schubert, T. Goll, N. Demmel, V. Usenko, J. Stueckler, and D. Cremers, "The TUM VI Benchmark for Evaluating Visual-Inertial Odometry," in *IROS*, October 2018.

[7] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g2o: A general framework for graph optimization," in *ICRA*, 2011, pp. 3607–3613.