2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

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

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

***Abstract*— We present Graphite, a GPU-accelerated nonlinear least squares graph optimization framework. It provides a CUDA C++ interface to enable the sharing of code between a real-time application, such as a SLAM system, and its optimization tasks. The framework supports techniques to reduce memory usage, including in-place optimization, support for multiple floating point types and mixed-precision modes, and dynamically computed Jacobians. We evaluate Graphite on well-known bundle adjustment problems and find that it achieves similar performance to MegBA, a solver specialized for bundle adjustment, while maintaining generality and using less memory. We also apply Graphite to global visual-inertial bundle adjustment on maps generated from stereo-inertial SLAM datasets, and observe speed-ups of up to $59\times$ compared to a CPU baseline. Our results indicate that our framework enables faster large-scale optimization on both desktop and resource-constrained devices.**

I. Introduction

Nonlinear optimization is a key component of many estimation problems in computer vision, graphics, and robotics. It is especially prevalent in keyframe-based simultaneous localization and mapping (SLAM) systems, which construct a representation of the environment while also determining a device's position and rotation. In particular, ORB-SLAM3 [1], a visual-inertial SLAM system, widely employs nonlinear optimization over hypergraphs for a broad array of frontend and backend tasks across its tracking, local mapping, and loop closing threads.

For real-time SLAM, it is necessary to perform these optimizations in a timely manner to remain up to date with changes in the device's state and surroundings. At the same time, the computational time and amount of memory needed to carry out each optimization may increase as the size of the map grows. This may be especially challenging for resource-constrained platforms when on-device SLAM is just one component of a larger system with several concurrent tasks.

Popular nonlinear optimizers [2], [3], [4] take advantage of multi-core CPU architectures, exploiting the inherent parallelism of sparse linear algebraic operations. However, some platforms used for SLAM [5] are equipped with on-board accelerators such as GPUs, which can be used to perform a variety of learning-based perception tasks efficiently, such as feature extraction, depth-estimation, object detection, and place recognition [6]. GPUs excel at these compute-heavy workloads due to their massively parallel architecture, which also makes them well-suited for nonlinear optimization.

To date, there have been several efforts to speed up existing optimization libraries by identifying and offloading expensive steps to GPUs. More recently, Ceres Solver [4] implemented GPU acceleration for the linear solver step of trust region optimization algorithms. Other work based on g^{2}o [3] focuses on efficiently reducing the size of the linear system to be solved [7] and accelerating numerical differentiation for computing Jacobian matrices [8]. Yet, while it is possible to improve the performance of existing libraries by offloading individual steps to the GPU, doing so introduces additional overhead from allocating GPU memory and transferring data between steps.

In contrast, new GPU-based frameworks have emerged which accelerate various steps of the optimization. Some use Python [9] or a domain-specific language (DSL) [10], [11] to allow users to write simple descriptions of nonlinear optimization problems that are used to automatically generate efficient solvers using complex code transformations. Others leverage machine learning frameworks to support batched optimization [12] and differentiable optimization [13], [14], [15] for training models for tasks such as feature extraction and feature matching. In addition, many solvers are specialized for specific classes of problems such as bundle adjustment [16], achieving large speedups.

However, existing GPU-accelerated solvers have several limitations which make them challenging to adapt for applications such as SLAM. They often model optimizable variables as simple numeric data types (e.g. a float, double, vector of doubles, etc.) or only provide data types for a specific optimization problem (e.g. a class representing an SE(3) transformation). Meanwhile, SLAM systems may represent variables as complex classes which consist of other classes. For example, an optimizable pose class may consist of multiple SE(3) transformations and camera references [1]. Existing solvers cannot represent these classes with a vector of numbers or predetermined data types because they cannot model their complex behaviours and data dependencies. Additionally, researchers develop real-time SLAM systems in a specific language of their choice, e.g., C++, making it difficult to use DSLs or Python-based optimizers, since they require optimizable data types and functionality to be faithfully reimplemented for another language. A third challenge is that GPU memory is often limited and shared between multiple tasks such as image processing and model inference. Moreover, existing libraries may demand additional GPU memory for operations on sparse matrices and linear solver workspace allocations [17]. Even worse, embedded platforms often have no dedicated GPU memory, so the CPU and GPU must compete for the same memory.

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

In this paper, we present Graphite, a GPU-accelerated nonlinear least squares graph optimization framework, which implements several techniques for balancing runtime performance and GPU memory usage, to enable large-scale optimization for desktop and embedded scenarios. It allows for mixed-precision solving using 64-bit, 32-bit, and 16-bit floating-point precisions, enabling faster and more memory-efficient optimization of graphs. To further reduce memory usage, the library supports dynamically computed Jacobians for matrix-free methods, as well as an automatic differentiation method with equivalent memory overhead to analytic differentiation. In addition, the iterative linear solver is aware of the structure of the graph, bypassing explicit sparse matrix formats and their associated memory costs. To support real-time SLAM and odometry, the framework provides a CUDA C++ interface, and uses a batching model which supports in-place optimization, allowing users to optimize GPU-accessible data without first transforming it into a solver-specific format. This avoids memory usage and runtime overhead from unnecessary copying, as well as explicit data transfers on platforms which share CPU and GPU memory.

To demonstrate the effectiveness and flexibility of our framework, we evaluate it on bundle adjustment problems, and find it performs comparably to MegBA [16] while using up to 78% less GPU memory. We also reimplement visual-inertial bundle adjustment on the GPU inside ORB-SLAM3, which can take hundreds of seconds on the CPU for sufficiently large problems, and demonstrate a speed up of up to $59\times$ on maps generated by processing stereo-inertial SLAM datasets [18], while using under 1 GiB of GPU memory on a desktop machine. The code is available at `https://github.com/sfu-rsl/graphite`.

To summarize our contributions:

- We design and implement a general mixed-precision optimization framework for on-device estimation problems which performs all major optimization steps on a GPU.
- We apply our framework to global visual-inertial bundle adjustment inside a SLAM system, which consists of 7 types of constraints and 5 types of variables.
- We evaluate our framework across different precisions and differentiation modes, using well-known bundle adjustment and SLAM datasets.

II. Background

In this section, we provide a brief discussion on nonlinear optimization over constraints, the Levenberg-Marquardt algorithm implemented by our framework, visual-inertial optimization, and relevant GPU programming concepts.

Optimizing Over Constraints: Some estimation problems can be modelled as an optimization over constraints. Each constraint represents a relationship between estimated parameters, an observation, and the error. The goal is then to find the parameters that minimize the total error across all constraints ($\mathcal{C}$). This may be formulated as

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

where x is the parameter vector, r_{ij} is the residual at x associated with a constraint between variables i and j and Σ_{ij} is the covariance matrix for the constraint. Alternatively, the quadratic form can be written as

$$\arg\min_x \frac{1}{2} r^\top \Sigma^{-1} r. \tag{2}$$

These problems may be expressed using an optimization framework [2], [3], [4] and solved using algorithms such as Powell's dog leg or Levenberg-Marquardt.

Levenberg-Marquardt for Nonlinear Least Squares: Given the nonlinear optimization problem in Eq. 1, the Levenberg-Marquardt algorithm (LM) can be used to iteratively find new parameters to drive the error towards a local minimum. LM modifies Gauss-Newton using a damping factor λ to achieve better convergence. It exploits the first-order Taylor expansion

$$f(x + \Delta x) \approx f(x) + \frac{\partial f(x)}{\partial x} \Delta x. \tag{3}$$

Each iteration then solves

$$\arg\min_{\Delta x} \frac{1}{2} ||r + J\Delta x||_\Sigma^2 + \frac{\lambda}{2} ||D\Delta x||_2^2 \tag{4}$$

at the current linearization point, where $J = \frac{\partial r}{\partial x}$, the Jacobian matrix, and $\frac{\lambda}{2}||D\Delta x||_2^2$ is a regularization term to control the step size where $D^\top D$ is $\text{diag}(J^\top \Sigma^{-1} J)$ or I. Taking the gradient of Eq. 4 with respect to Δx, setting it to 0, and rearranging the result gives the normal equations

$$(J^\top \Sigma^{-1} J + \lambda D^\top D)\Delta x = -J^\top \Sigma^{-1} r. \tag{5}$$

This can be more simply rewritten in terms of the regularized Hessian approximation H_λ and gradient b as

$$H_\lambda \Delta x = -b. \tag{6}$$

Solving this linear system in each iteration yields Δx, which is a candidate parameter step to reduce the total error. Here, H_λ is a sparse symmetric positive definite block matrix [19].

Visual-Inertial Optimization: Given a set of keyframes κ and l map points, the goal of visual bundle adjustment is to find new parameters that better satisfy camera reprojection error constraints. The cost may be formulated as Eq. 7 [1]

$$\sum_{j=0}^{l-1} \sum_{i \in \kappa j} \rho_{\text{Huber}}(\|r_{ij}\|_{\Sigma_{ij}}^2) \tag{7}$$

where r_{ij} is the residual vector of the reprojection constraint between observed map point j and keyframe i. In other words, r_{ij} is the difference, in 2D coordinates, between the computed camera projection of a map point in a keyframe image using current parameters and its measured observation. To reduce sensitivity to feature mismatches between keyframes and potential outliers, a robust Huber kernel ρ_{Huber} is used [1]. Visual-inertial bundle adjustment introduces additional terms to the optimization for inertial parameters. For k+1 keyframes, this is represented as [1]

$$\sum_{i=1}^{k} \|r_{I_{i-1,i}}\|_{\Sigma_{I_{i-1,i}}}^2 + \sum_{i=1}^{k} \|r_{b_{i-1,i}}\|_{\Sigma_{b_{i-1,i}}}^2 \tag{8}$$

where $r_{I_{i-1,i}}$ is the inertial residual between consecutive keyframes, representing the discrepancy between preintegrated IMU measurements and the change computed from keyframe parameters. These parameters include rotation, velocity, and position. Similarly, $r_{b_{i-1,i}}$ is the accelerometer and gyroscope bias residual between consecutive keyframes.

GPU Programming: Modern GPUs are based on massively parallel Single-Instruction, Multiple-Thread (SIMT) architecture, where threads execute common instructions in groups. CUDA [17] is a programming model that enables writing specialized functions, called *kernels*, which can be dispatched onto a GPU in order to process a batch of similar tasks in parallel. CUDA uses a hierarchical execution model where threads are grouped into 3D thread blocks. Thread blocks are further organized into a grid. When a kernel is launched, threads and blocks are assigned identifiers which can be queried by a task to decide inputs, outputs, and behaviour. Threads can then communicate with each other using shared memory and built-in functions. Additionally, multiple kernels can be launched in different streams, where each stream is a sequence of commands, to allow their executions to overlap on the GPU.

III. Related Work

As our work proposes a GPU-based approach for nonlinear least squares optimization for problems such as bundle adjustment, we review literature on these topics.

Although there are several versatile libraries for least squares optimization in robotics and computer vision [2], [3], [4], [20], [21], [22], they are primarily CPU-based and generally limit GPU acceleration to the linear solver stage [4]. Our framework differs by performing the entire optimization on the GPU, to enable greater parallelization for variable and constraint operations and avoid data transfers between intermediate steps to improve performance.

There are also a number of GPU-accelerated solvers based on domain-specific languages (DSLs) [9], [10], [11] or Python machine learning frameworks [12], [13], [14], [15]. However, these approaches may result in code duplication, lack the expressiveness needed to model complex types, or involve explicit language interoperation, depending on the application. Our framework avoids this by allowing problems to be described directly using user-defined data types, preventing code duplication, and enabling easier prototyping.

Several prior works also explore specialized implementations for bundle adjustment on the GPU, applying direct or iterative solving techniques [16], [23], [24], [25], [26], [27], [28]. While performant, these works target a specific type of visual bundle adjustment which only uses camera poses, 3D points, and binary constraints. Meanwhile, SLAM and odometry systems often add several new constraints by incorporating inertial information [1], [29], where some constraints have as many as six variables. They may also perform other optimization tasks such as motion-only bundle adjustment and pose graph optimization [1], [29]. Our framework resolves this limitation by supporting user-defined data types along with unary, binary, and n-ary GPU constraints.

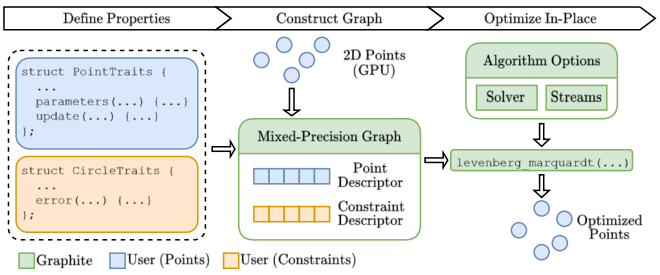

Fig. 1. An example Graphite workflow. The user first defines properties for their variables and constraints, then constructs a graph by creating descriptors, and finally optimizes their variables using Levenberg-Marquardt.

While prior GPU-accelerated solvers utilize code generation and leverage existing machine learning frameworks, we adopt a design inspired by libraries such as g^{2}o and Ceres Solver, to support the level of functionality needed for visual-inertial SLAM [1] and odometry systems [29], discussed further in Section V. The CUDA C++ interface enables the ability to leverage existing data structures within these systems for optimization. This approach also enables in-place optimization on platforms with full unified memory support, so that variables need not be explicitly transferred. Additionally, we support the ability to mix higher and lower floating-point precisions to reduce GPU memory usage.

IV. Design

Graphite implements nonlinear least squares graph optimization on the GPU, based on the Levenberg-Marquardt algorithm (Section II). Its design is guided by the need to minimize the amount of time required for optimization while also using as little GPU memory as possible, which is crucial for resource-constrained systems. An optimization problem is represented by a graph constructed from a set of descriptors, which are collections of optimizable vertices (variables) and unary, binary, and n-ary constraints. Descriptors provide a batching mechanism for the GPU and allow vertices and constraints to be represented with user data types that are compatible with CUDA device code (Section IV-A). This avoids the need to first transform variables into a solver-specific format for optimization, a step which is often required by other libraries. As is standard with other frameworks, Graphite supports fast analytic differentiation (Section IV-B) and convenient automatic differentiation (Section IV-C) for computing Jacobian matrices used by the optimization algorithm. To perform the optimization process quickly and in a memory-efficient manner, Graphite also supports mixed-precision solving (Section IV-D).

We show a running example of the workflow in Figure 1 for a 2D point optimization problem, where a user adjusts 2D points stored in GPU-accessible memory to lie on the boundary of a circle. They first define static properties for their 2D points and circle equation constraints. Next, they create the corresponding descriptors and construct a graph. Lastly, the user passes the graph to the Levenberg-Marquardt algorithm along with desired options, optimizing the points in-place. We discuss this process with concrete examples next.

A. Descriptor Batching Model

A typical approach to offloading work onto a GPU is to organize similar tasks into batches, and process tasks within a batch concurrently across numerous threads in order to take advantage of the massively parallel architecture. However, existing graph-based optimizers store different types of vertices and constraints indiscriminately together in the graph. This is not ideal for batching work on the GPU, because each thread processing an item (a vertex or constraint) may branch and execute a different code path, which is detrimental to performance [17]. Therefore, to enable a GPU-friendly approach for graph-based optimization, Graphite allows users to construct an optimizable graph from collections of vertices and constraints of the same type. This enables GPU threads to process batches of identical items, which minimizes branching. We represent these collections using descriptors, which we describe next with our point optimization example.

We demonstrate how the user models the point optimization problem from Figure 1 in Graphite, where the user selects single precision for the variables and constraints, and 16-bit brain floating point for the linear system.

First, the user defines static properties for their vertices (points) and constraints (based on the circle equation), which are necessary for creating descriptors, as shown in Figure 1. At compile time, these properties are used to specialize each CUDA kernel, so that each thread carries out the same type of computation, improving performance.

For a vertex descriptor, these properties include information such as the underlying data type (class), the parameter block size, a method to get its parameter block, and a method to update each vertex for a new step. Listing 1 shows how the user defines the properties for a 2D point descriptor. They specify the underlying vertex data type as a 2×1 Eigen [30] matrix (Line 2, Line 7), the length of the parameter block as 2 (Line 6), the method to get the parameter block (Lines 9 to 13), and the update method (Lines 15-18).

```
1  // Point definition
2  template <typename T> using Point = Eigen::Matrix<T, 2, 1>;
3
4  // Traits for Point
5  template <typename T, typename S> struct PointTraits {
6    static constexpr size_t dimension = 2;
7    using Vertex = Point<T>;
8
9    template <typename P>
10   d_fn static void parameters(const Vertex &vertex, P *parameters) {
11     Eigen::Map<Eigen::Matrix<P, dimension, 1>> params_map(parameters);
12     params_map = vertex.template cast<P>();
13   }
14
15   d_fn static void update(Vertex &vertex, const T *delta) {
16     Eigen::Map<const Eigen::Matrix<T, dimension, 1>> d(delta);
17     vertex += d;
18   }
19 };
20
21 // Type alias
22 template <typename T, typename S>
23 using PointDescriptor = VertexDescriptor<T, S, PointTraits<T, S>>;
```

Listing 1. Properties of a vertex descriptor for a 2D point.

Next, the user defines static properties for their constraints, including which vertex descriptors are involved, the data types (classes) of the observations and any non-optimizable data, the loss function used, and the differentiation mode (discussed in Section IV-B and Section IV-C). These properties also specify how to compute the residual for a constraint, and how to compute the Jacobians when using analytic differentiation (Section IV-B). Listing 2 shows this for the point optimization problem. The user specifies a residual of size 1 (Line 3), specifies the vertex descriptor and data types (Lines 4-6), chooses the default loss function (Line 7), and chooses to use automatic differentiation (Line 8), so no Jacobian evaluation function is specified. Additionally, Lines 10-16 tell the descriptor how to compute the residual based on the equation of a circle.

```
1  // Traits for the circle constraint
2  template <typename T, typename S> struct CircleTraits {
3    static constexpr size_t dimension = 1;
4    using VertexDescriptors = std::tuple<PointDescriptor<T, S>>;
5    using Observation = T;
6    using Data = Empty; // unused - not passed to error function
7    using Loss = DefaultLoss<T, dimension>;
8    using Differentiation = DifferentiationMode::Auto;
9
10   template <typename D>
11   d_fn static void error(const D *point, const T &obs, D *error) {
12     const auto x = point[0]; // Note: A const Point<T>& can also be passed in.
13     const auto y = point[1]; // The framework automatically determines
14     const auto r = obs;      // how to call the user's function.
15     error[0] = x * x + y * y - r * r;
16   }
17 };
18
19 template <typename T, typename S>
20 using CircleDescriptor = FactorDescriptor<T, S, CircleTraits<T, S>>;
```

Listing 2. Properties for the circle equation constraint descriptor.

We specify properties of descriptors in this way because it lets users declare the behaviour of their types and automatically generate optimized GPU code, without relying on runtime lookups (as observed in some CPU-based optimizers [3]) to decide how each item in the graph is processed. Following Figure 1, after defining these properties, the user creates descriptors in order to construct an optimizable graph.

The user first creates a vertex descriptor for the collection of points, as shown in Listing 3. Before doing so, they declare a graph with the desired precisions in Lines 1-3. Then in Lines 4-6, they create the vertex descriptor, reserve memory for the desired number of vertices, and add the descriptor to the graph. Then from Lines 8-10, they add a pointer for each vertex to the descriptor, and associate it with an identifier. In creating this descriptor, the user defines a batch of vertices which is processed efficiently across many GPU threads.

```
1  using FP = float; // vertex, constraint precision
2  using SP = __nv_bfloat16; // linear system precision
3  Graph<FP, SP> graph; // Declare graph
4  auto point_desc = PointDescriptor<FP, SP>(); // Create descriptor
5  point_desc.reserve(num_vertices); // Reserve memory for num_vertices
6  graph.add_descriptor(&point_desc); // Add to graph
7
8  for (size_t vertex_id = 0; vertex_id < num_vertices; ++vertex_id) {
9      point_desc.add_vertex(vertex_id, &points[vertex_id]); // Add points
10 }
```

Listing 3. Creating a vertex descriptor for 2D points.

Next, they create a descriptor for the constraints in Listing 4. As in the previous example, they initialize the descriptor, reserve memory for the desired number of constraints (one constraint for each vertex), and add the descriptor to the graph (Lines 1-3). They initialize the default loss function on Line 4, and then create the desired number of constraints (Lines 6-8). When creating each constraint, the user specifies the identifiers of the vertices involved, the observation (the radius of the circle), the precision matrix (`nullptr` defaults to the identity matrix), optional constraint data (unused), and the loss function parameters. Again, by constructing this descriptor, the user enables each constraint to be processed concurrently and efficiently on the GPU.

```
auto circle_desc = CircleDescriptor<FP, SP>(&point_desc); // Create descriptor
circle_desc.reserve(num_vertices); // Reserve memory for constraints
graph.add_descriptor(&circle_desc); // Add descriptor to graph
const auto loss = DefaultLoss<FP, 1>(); // loss function

for (size_t vertex_id = 0; vertex_id < num_vertices; ++vertex_id) {
    circle_desc.add_factor({vertex_id}, radius, nullptr, Empty(), loss);
}
```

Listing 4. Creating a descriptor for constraints.

After the user finishes constructing the graph, they pass it to an optimization algorithm, along with various options, including the desired linear solver (discussed in Section IV-D), optimizing the points in-place (Figure 1). For the point optimization example, this is shown in Listing 5. First, the user decides to fix the value of the last point they created (Line 1) and decides to disable the constraint which was created for the third point by changing its level to 1, as in g^{2}o [3] (Line 2). Next, the user configures an iterative linear solver, by selecting an identity matrix for the preconditioner (Line 3) and then instantiating the solver (Line 4) with their desired parameters. They create one stream (Line 5), since each constraint only has one vertex. Lastly, they configure the Levenberg-Marquardt algorithm on Lines 7-12, and run the optimization on Line 13.

```
point_desc.set_fixed(num_vertices - 1, true); // Set the last vertex as fixed
circle_desc.set_active(2, 0x1); // Disable third constraint for point 2
IdentityPreconditioner<FP, SP> preconditioner;
PCGSolver<FP, SP> solver(50, 1e-6, 10.0, &preconditioner);
StreamPool streams(1); // chose 1 stream for 1 vertex per constraint

optimizer::LevenbergMarquardtOptions<FP, SP> options;
options.solver = &solver;
options.initial_damping = 1e-6;
options.iterations = 10;
options.optimization_level = 0;
options.streams = &streams;
optimizer::levenberg_marquardt<FP, SP>(&graph, &options);
```

Listing 5. Example of optimizing noisy 2D points.

B. Analytic Differentiation

In graph-based optimization, constraints may involve one or more vertices, each of which generate a Jacobian matrix. A typical way to compute Jacobians quickly is by evaluating an analytic (closed-form) expression. Graphite allows Jacobians to be evaluated directly using analytic expressions defined in a user-written function. These expressions are derived by hand or generated using a symbolic toolkit [22], [31].

A typical approach on the CPU is to compute each matrix for a constraint sequentially [3], [4]. However, to take advantage of the many GPU threads available, Graphite attempts to compute each Jacobian concurrently in separate streams (described in Section II). Additionally, Graphite allows these Jacobians to be evaluated dynamically [23], at the point of computation (e.g. matrix-vector multiplication), to enable matrix-free solving methods which use less memory.

C. Automatic Differentiation

Another method to compute Jacobians is through forward mode automatic differentiation. For example, Ceres Solver computes the gradient of a residual function using dual numbers in the form $a + b\varepsilon$, where a value is represented by real component a, and the partial derivative is represented by infinitesimal component b, where $\varepsilon^2 = 0$ [4]. To extend this for multiple inputs, Ceres implements a data structure called a *Jet*, which replaces the infinitesimal component with a vector [4]. Ren et. al [16] extend this concept to the GPU by proposing *JetVectors*, which adopt a structure of arrays data layout to enable coalesced memory transactions for better performance. However, this design is intended for optimization across multiple devices and may require more GPU memory than available on a single device [16].

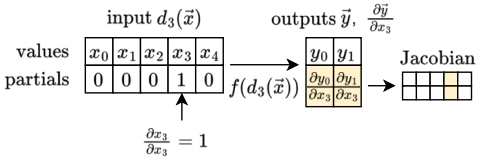

Fig. 2. A CUDA thread calculates $\frac{\partial f}{\partial x_3}(\vec{x})$ by replacing the input with a vector of dual numbers and writes it to the corresponding Jacobian column.

To handle environments with less available memory, we avoid allocating storage for dual numbers entirely. Instead of using Jets or JetVectors, when computing the residual of a constraint, each kernel replaces the input parameters with vectors of dual numbers, one for each vertex. Each CUDA thread then calculates the column of a Jacobian matrix (Figure 2), by computing the residual using these vectors. The column to be computed is changed by setting one partial to 1, and the others to 0. As shown in Section V, this uses the same amount of memory as analytic differentiation, allowing Jacobians to be computed efficiently and conveniently without first needing to derive an expression.

D. Mixed-Precision Solver

To enable mixed-precision solving, which allows the optimization to use less memory, Graphite supports the customization of floating point precisions for vertices and the linear system (Eq. 6). Essentially, these two precisions can be customized according to trade-offs between convergence, runtime performance, and memory usage. A typical approach would be to use a higher precision (double or single precision) for the vertices to maintain numerical stability, and an equal or lower precision (single precision or brain floating point) for the linear system to reduce the amount of memory needed to store it [32]. In contrast, existing solvers support only one precision at a time [16], or only allow casting the linear system to single precision [4]. As shown in Section V, using lower precisions greatly reduces the overall optimization time and the amount of GPU memory required.

As described in Section II, each iteration of Levenberg-Marquardt optimization computes a parameter step Δx. To compute this quickly on the GPU, we apply preconditioned conjugate gradients (PCG) on the Hessian, with implicit evaluation using the Jacobian matrices [23], since this does not require storing the Hessian, and comprises of algebraic operations which are largely amenable to parallelization. However, directly using PCG with lower precision types can result in poor convergence. To mitigate this, we adopt techniques which are known to help with solving larger systems. These include clamping values along the Hessian diagonal [4], rescaling the columns of each Jacobian block according to the Hessian diagonal [4], [32], and normalizing the PCG residual in each iteration [33].

In applying PCG, to improve convergence, we also use

TABLE I

BUNDLE ADJUSTMENT RESULTS FOR BAL (DESKTOP), INCLUDING MEAN SQUARED ERROR, TIME (SECONDS), AND GPU MEMORY (MIB).

BAL Problem	Problem Size[1]	Evaluation Metric	Graphite FP64	Graphite FP32	Graphite FP32-BF16	Ceres CGNR	MegBA FP64	MegBA FP32	DeepLM PCG
Dubrovnik-16	16 Img	MSE	0.43	0.43	0.43	0.43	0.43	0.43	0.43
	22106 Pts	Time	0.22	0.11	0.12	1.64	0.17	0.16	3.88
	83718 Obs	GPU Memory	298	282	278	336	406	352	814
Trafalgar-21	21 Img	MSE	1.67	1.67	1.68	1.67	1.67	1.67	1.67
	11315 Pts	Time	0.11	0.069	0.068	0.82	0.098	0.088	1.61
	36455 Obs	GPU Memory	280	274	270	318	326	298	764
Ladybug-49	49 Img	MSE	0.85	0.85	0.85	0.84	0.84	0.84	0.84
	7776 Pts	Time	0.063	0.041	0.046	0.66	0.086	0.074	1.14
	31843 Obs	GPU Memory	278	274	270	316	326	300	740
Venice-1778	1778 Img	MSE	0.67	0.67	0.67	0.67	0.66	0.67	0.67
	993923 Pts	Time	6.00	2.80	2.87	43.71	6.96	6.15	41.67
	5001946 Obs	GPU Memory	1944	1190	950	2160	7810	4528	5708
Final-4585	4585 Img	MSE	1.13	1.13	1.14	1.14	1.12	1.13	1.13
	1324582 Pts	Time	9.62	4.69	3.75	73.49	13.19	9.82	43.11
	9125125 Obs	GPU Memory	3102	1822	1386	3546	13882	7960	7158
Final-13682	13682 Img	MSE	1.50	1.50	1.50	1.51	OOM	OOM	1.51
	4456117 Pts	Time	26.60	11.65	11.67	210.64	OOM	OOM	55.62
	28987644 Obs	GPU Memory	9298	5212	3828	10640	OOM	OOM	15044

[1] Shows the number of camera images (Img), points (Pts), and observations (Obs).

TABLE II

BUNDLE ADJUSTMENT ACROSS DIFFERENTIATION MODES, INCLUDING MEAN SQUARED ERROR, TIME (SECONDS), AND GPU MEMORY (MIB).

Problem	Metric	Analytic	Dynamic	Auto
Final-13682 FP64	MSE	1.50	1.50	1.50
	Time	26.60	417	38.81
	GPU Memory	9298	3988	9298
Final-13682 FP32	MSE	1.50	1.50	1.50
	Time	11.65	25.22	11.46
	GPU Memory	5212	2556	5212
Final-13682 FP32-BF16	MSE	1.50	1.50	1.50
	Time	11.67	24.48	11.27
	GPU Memory	3828	2500	3828

a block Jacobi preconditioner [19], because it is relatively simple to compute and does not require much memory due to its sparse block-diagonal matrix structure. We have observed that storing this preconditioner at too low of a precision can also result in convergence issues. Thus, when a 16-bit precision type such as bfloat16 is specified for the linear system, we compute and store this preconditioner at single or double precision, according to the precision of the variables.

Additionally, since the PCG solver does not require storing the Hessian matrix, we avoid explicitly constructing sparse matrices for the optimization problem. Instead, when performing an operation such as matrix-vector multiplication, the kernel uses the structure of the graph itself. This eliminates additional memory and runtime costs associated with converting and storing matrices in different formats, although each kernel must now be aware of the graph structure and whether a variable is fixed or a constraint is inactive.

V. EVALUATION

We evaluate Graphite in terms of the error reduction, runtime performance, and GPU memory usage. First, we compare the solver in different precision modes, using small to large problems from Bundle Adjustment in the Large (BAL) [19], which serves as a common benchmark across optimization libraries. Next, we compare analytic, dynamic, and automatic differentiation modes. Lastly, we run full-inertial bundle adjustment with Graphite inside ORB-SLAM3, using maps generated from outdoors sequences in TUM-VI [18], a well-known dataset for evaluating visual-inertial SLAM systems. To assess Graphite in high-performance and resource-constrained environments, we run these experiments on a desktop machine with a 12-core Intel Core i7-12700K CPU at 3.8-4.9 GHz, 10496-core NVIDIA RTX 3090 GPU at 1.9 GHz, and 64 GB RAM, and also on an NVIDIA Jetson Orin Nano with a 6-core ARM Cortex-A78AE CPU at 1.7 GHz, 1024-core NVIDIA Ampere GPU at 1.0 GHz, and 8 GB RAM. We use unified memory to store optimizable variables.

A. Mixed-Precision Modes

We evaluate Graphite on BAL datasets using double precision (FP64), single precision (FP32), and single precision mixed with bfloat16 (FP32-BF16) on the desktop machine. We use bfloat16 rather than 16-bit half precision, because it can approximately represent the same range as single precision. For comparison, we select two general GPU-accelerated optimization libraries with BAL support and similar conjugate gradient solvers, Ceres Solver [4] and DeepLM [12], as well as MegBA [16], a fully-accelerated, high-performance bundle adjuster with single and double precision modes. Where available, we use analytic Jacobians, which have the least runtime overhead. We generate Jacobian expressions for Graphite using wrenfold [31]. To allow each optimizer sufficient opportunity to converge, we use a maximum of 50 iterations. For PCG in MegBA, the maximum number of iterations is set to 100, while 10 is used for the other configurations, since they solve the full Hessian system rather than a reduced one, which involves more work per iteration. As shown in Table I for single run experiments, Graphite uses substantially less memory when using BF16 to store the Jacobian matrices, at the cost of slightly worse convergence. For runtime, FP64 takes the longest, while FP32 and FP32-BF16 take a comparable amount of time. Compared to similar libraries, Graphite takes less time since it accelerates constraint-specific calculations as well. Although MegBA achieves slightly lower MSE in a few instances, Graphite uses $\frac{1}{4}$ the memory for large problems, since it avoids storing

TABLE III

FULL-INERTIAL BUNDLE ADJUSTMENT ON ORB-SLAM3 MAPS, INCLUDING THE TOTAL χ^2 ERROR, TIME (SECONDS) AND GPU MEMORY (MIB).

TUM-VI Sequence	Initial χ^2 Error	Graph Size[1]	Evaluation Metric[2]	g^{2}o LDLT	g^{2}o PCG	Graphite FP64	Graphite FP32	Graphite FP32-BF16
outdoors1 Desktop	1.0803×10^6	3131 KF 58886 MP 348548 Co	Final χ^2 Time GPU Mem.	3.0918×10^5 1424.40 —	4.4965×10^5 224.08 —	4.3850×10^5 4.96 678	4.6842×10^5 1.71 580	5.9817×10^5 1.18 542
outdoors2 Desktop	7.3020×10^5	2032 KF 57641 MP 338364 Co	Final χ^2 Time GPU Mem.	2.9105×10^5 698.68 —	3.5100×10^5 147.57 —	3.5012×10^5 2.80 656	3.4090×10^5 3.56 564	4.8966×10^5 1.10 530
outdoors3 Desktop	6.4905×10^5	1595 KF 36536 MP 272307 Co	Final χ^2 Time GPU Mem.	2.5617×10^5 2957.40 —	3.0916×10^5 99.69 —	3.0354×10^5 1.69 636	3.0684×10^5 1.32 562	3.3927×10^5 1.13 534
outdoors1 Orin Nano	7.1589×10^5	2483 KF 48621 MP 278661 Co	Final χ^2 Time GPU Mem.	2.2494×10^5 2411.40 —	2.9363×10^5 512.58 —	2.9112×10^5 122.61 5673, 306	2.9348×10^5 93.10 5462, 95	3.6397×10^5 27.90 5433, 66
outdoors2 Orin Nano	7.3551×10^5	1972 KF 33831 MP 213125 Co	Final χ^2 Time GPU Mem.	2.1432×10^5 1141.40 —	3.3585×10^5 345.46 —	3.3731×10^5 53.63 5128, 328	3.5705×10^5 32.58 5031, 231	3.5889×10^5 29.73 5017, 217
outdoors3 Orin Nano	5.3520×10^5	1589 KF 24524 MP 174070 Co	Final χ^2 Time GPU Mem.	1.6574×10^5 2166.1 —	1.9508×10^5 355.34 —	1.9439×10^5 93.71 4674, 282	2.0736×10^5 33.53 4473, 81	2.1015×10^5 29.64 4449, 57

[1] Shows the number of keyframes (KF), map points (MP), and constraints (Co). Each keyframe generates four variables (pose, velocity, IMU biases).
[2] For the Orin Nano, we report the peak GPU memory usage and peak usage minus the baseline usage before Graphite runs.

Hessian matrices, and receives a greater speedup when using single precision. These results highlight the runtime and memory advantages of performing the entire optimization on the GPU and directly utilizing the graph structure to avoid constructing intermediate matrices.

B. Differentiation Modes

We report Graphite's performance using analytic, dynamic, and automatic differentiation modes in Table II for Final-13682, the largest dataset in BAL. Automatic differentiation allocates the same amount of memory as analytic differentiation, taking approximately the same time at lower precisions, and significantly more at FP64. Meanwhile, the dynamic mode requires the least amount of GPU memory since Jacobians are not stored, although precision matrices are still used. While the dynamic mode exhibits a larger slowdown for FP64, this can be mitigated by switching to lower precisions. Our results show that automatic differentiation can be achieved with relatively low memory and runtime overhead for lower precisions, enabling rapid prototyping of optimizable constraints, while dynamically computed Jacobians enable more memory efficient solving at the cost of higher runtimes.

C. Case Study: Visual-Inertial Bundle Adjustment

We apply Graphite to full-inertial bundle adjustment in ORB-SLAM3, which introduces several additional variables and constraints to visual bundle adjustment (Section II).

Implementation: To use Graphite for visual-inertial bundle adjustment, we reimplement vertices and constraints involved in the optimization, external to the library. The original data types in ORB-SLAM3 cannot be used as-is, since some contain extraneous data, involve dynamic memory allocations which must be manually transferred from the CPU to the GPU, or implement methods which are not suitable for GPU execution (e.g. a method which accesses a resource shared between different CPU threads). Therefore, we implement lightweight template data types as a replacement, which contain only the subset of data and functionality needed for visual-inertial optimization. These include data types for the combined IMU camera pose vertex, the cameras (pinhole and fisheye), and the preintegrated IMU measurements. The remaining vertex data types for the map points, velocities, gyro biases, and accelerometer biases are 3×1 vectors, so we directly represent these using Eigen [30]. To use these data types with Graphite, we define descriptors, as in Listing 1.

To model the optimization constraints, we create eight descriptors for monocular and stereo reprojection errors, the inertial residual (with and without Huber loss), gyro and accelerometer bias residuals, and bias priors. Following the original implementation, we use the same expressions for computing the residuals and Jacobians of each constraint along with Sophus [34] for Lie groups. When constructing the graph, to avoid overhead from reallocating memory, we predetermine the size of the optimization and reserve the required memory for descriptors and optimizable variables.

Experiment: Next, we evaluate the performance of visual-inertial bundle adjustment inside ORB-SLAM3 using Graphite. Among compatible datasets, we choose outdoors sequences from the TUM-VI dataset, since they generate large maps consisting of thousands of keyframes. Within ORB-SLAM3, we perform full (global) visual-inertial bundle adjustment on each of the generated maps. For the baseline, we compare Graphite to the existing LDLT-based g^{2}o implementation, using the original LM algorithm without modified termination criteria [1] for more consistent behaviour. We also include a PCG baseline, since our implementation uses PCG. Note that the challenges discussed in Section III restrict which solvers can be used. Limiting factors include language compatibility and sufficient support for user data types involved in complex constraints (e.g. with up to six variables). For all solvers, the parameter recovery step is skipped so that all optimizations run on the same map. The optimization is set to run for up to 100 iterations. Block ordering is enabled for LDLT as we found this improves the

runtime performance. For g^{2}o PCG, we use a maximum of 50 iterations, with 1×10^{-6} absolute tolerance, while for our solver, we use a maximum of 100 iterations, with 1×10^{-12} absolute tolerance, and a rejection ratio of 10. The reason for different termination criteria is that g^{2}o computes and solves a reduced system, while Graphite uses the full Hessian and gradient. Both solvers apply block Jacobi preconditioning.

Results: The results for TUM-VI are given in Table III. The LDLT baseline converges to the lowest χ^2 error for all maps due to computing the exact parameter step at each iteration, at the cost of extremely high runtimes. Compared to the PCG baseline, we achieve a relative speedup of up to $59\times$ on the desktop and up to $6\times$ on the Orin Nano, despite using the full Hessian. This difference in speedup is in part due to the Orin Nano having far fewer CUDA cores and operating at lower clock speeds than its desktop counterpart. Moreover, while lower precision modes take less time on the Jetson, FP64 generally achieves lower χ^2 values and runs for more iterations. Note that the Jetson reports higher memory usages due to the different architecture, so we also include the peak usage minus the baseline usage before Graphite first runs. Overall, Graphite achieves a significant reduction in time, enabling faster large-scale visual-inertial optimization.

VI. Conclusion

We have implemented a general GPU-accelerated nonlinear optimization framework based on Levenberg-Marquardt for sparse estimation problems in CUDA C++, which enables the sharing of data types and functions between real-time applications and optimization tasks. Graphite achieves a relatively lower memory footprint by supporting in-place optimization, inferring matrix structures from the optimizable graph, supporting mixed-precision solving, and enabling matrix-free methods. Our results show that our framework enables large-scale optimization over complex constraints for desktop and resource-constrained devices. In the future, we would like to support more algorithms and solving techniques.

References

[1] C. Campos, R. Elvira, J. J. Gómez, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM3: An accurate open-source library for visual, visual–inertial, and multimap slam," *T-RO*, vol. 37, no. 6, 2021.

[2] F. Dellaert and GTSAM Contributors, "borglab/gtsam," May 2022. [Online]. Available: https://github.com/borglab/gtsam

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

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

[5] J. Jeon, S. Jung, E. Lee, D. Choi, and H. Myung, "Run Your Visual-Inertial Odometry on NVIDIA Jetson: Benchmark Tests on a Micro Aerial Vehicle," *RA-L*, vol. 6, 2021.

[6] S. Mokssit, D. B. Licea, B. Guermah, and M. Ghogho, "Deep learning techniques for visual slam: A survey," *IEEE Access*, vol. 11, 2023.

[7] 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 *ICRA*, 2023.

[8] D. Kumar, S. Gopinath, K. Dantu, and S. Y. Ko, "JacobiGPU: GPU-Accelerated Numerical Differentiation for Loop Closure in Visual SLAM," in *ICRA*, May 2024.

[9] C. Wefelscheid and O. Hellwich, "OpenOF - Framework for Sparse Non-linear Least Squares Optimization on a GPU," in *VISAPP*, INSTICC. SciTePress, 2013, pp. 260–267.

[10] Z. Devito, M. Mara, M. Zollhöfer, G. Bernstein, J. Ragan-Kelley, C. Theobalt, P. Hanrahan, M. Fisher, and M. Niessner, "Opt: A domain specific language for non-linear least squares optimization in graphics and imaging," *TOG*, vol. 36, 10 2017.

[11] M. Mara, F. Heide, M. Zollhöfer, M. Nießner, and P. Hanrahan, "Thallo – Scheduling for High-Performance Large-Scale Non-Linear Least-Squares Solvers," *TOG*, vol. 40, 10 2021.

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

[13] B. Yi, M. Lee, A. Kloss, R. Martín-Martín, and J. Bohg, "Differentiable factor graph optimization for learning smoothers," in *IROS*, 2021.

[14] L. Pineda, T. Fan, M. Monge, S. Venkataraman, P. Sodhi, R. T. Chen, J. Ortiz, D. DeTone, A. Wang, S. Anderson, J. Dong, B. Amos, and M. Mukadam, "Theseus: A Library for Differentiable Nonlinear Optimization," *NeurIPS*, 2022.

[15] C. Wang, D. Gao, K. Xu, J. Geng, Y. Hu, Y. Qiu, B. Li, F. Yang, B. Moon, A. Pandey, Aryan, J. Xu, T. Wu, H. He, D. Huang, Z. Ren, S. Zhao, T. Fu, P. Reddy, X. Lin, W. Wang, J. Shi, R. Talak, K. Cao, Y. Du, H. Wang, H. Yu, S. Wang, S. Chen, A. Kashyap, R. Bandaru, K. Dantu, J. Wu, L. Xie, L. Carlone, M. Hutter, and S. Scherer, "PyPose: A library for robot learning with physics-based optimization," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[16] 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.

[17] NVIDIA, "CUDA Toolkit Documentation," 2024. [Online]. Available: https://docs.nvidia.com/cuda/

[18] 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.

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

[20] 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," *IJRR*, vol. 38, no. 2-3, pp. 95–125, 2019. [Online]. Available: https://doi.org/10.1177/0278364918784361

[21] J. Solà, J. Vallvé, J. Casals, J. Deray, M. Fourmy, D. Atchuthan, A. Corominas-Murtra, and J. Andrade-Cetto, "WOLF: A Modular Estimation Framework for Robotics Based on Factor Graphs," *RA-L*, vol. 7, no. 2, Apr. 2022.

[22] H. Martiros, A. Miller, N. Bucki, B. Solliday, R. Kennedy, J. Zhu, T. Dang, D. Pattison, H. Zheng, T. Tomic, P. Henry, G. Cross, J. VanderMey, A. Sun, S. Wang, and K. Holtz, "SymForce: Symbolic Computation and Code Generation for Robotics," in *RSS*, 2022.

[23] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, "Multicore bundle adjustment," in *CVPR*. IEEE Computer Society, 2011.

[24] R. Hänsch, I. Drude, and O. Hellwich, "Modern methods of bundle adjustment on the gpu," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. III-3, 6 2016.

[25] Fixstars Corporation, "Cuda bundle adjustment," 2022. [Online]. Available: https://github.com/fixstars/cuda-bundle-adjustment

[26] M. Cao, L. Zheng, W. Jia, and X. Liu, "Fast incremental structure from motion based on parallel bundle adjustment," vol. 18. Springer Science and Business Media Deutschland GmbH, 4 2021.

[27] T. Fan, J. Ortiz, M. Hsiao, M. Monge, J. Dong, T. Murphey, and M. Mukadam, "Decentralization and acceleration enables large-scale bundle adjustment," *arXiv:2305.07026*, 2023.

[28] H. Han and H. Yang, "Building rome with convex optimization," no. arXiv:2502.04640, July 2025, arXiv:2502.04640 [cs].

[29] T. Qin, P. Li, and S. Shen, "VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator," *T-RO*, vol. 34, 8 2018.

[30] G. Guennebaud, B. Jacob, *et al.*, "Eigen," https://libeigen.gitlab.io, 2010.

[31] G. Cross, "wrenfold: Symbolic code generation for robotics," *Journal of Open Source Software*, vol. 10, no. 105, 2025.

[32] T. Qin and S. Dayal, "Low-precision arithmetic makes robot localization more efficient," Apr 2023. [Online]. Available: https://www.amazon.science/blog/low-precision-arithmetic-makes-robot-localization-more-efficient

[33] Y. Guo, E. de Sturler, and T. Warburton, "An adaptive mixed precision and dynamically scaled preconditioned conjugate gradient algorithm," no. arXiv:2505.04155, May 2025, arXiv:2505.04155 [math].

[34] H. Strasdat and S. Lovegrove, "Sophus," 2018. [Online]. Available: https://github.com/strasdat/Sophus