ORIANNA: An Accelerator Generation Framework for Optimization-based Robotic Applications

Yuhui Hao*
yuhuihao@tju.edu.cn
Tianjin University
China

Yiming Gan*
ygan10@ur.rochester.edu
ICT, Chinese Academy of Sciences
China

Bo Yu
inzaghi1984@gmail.com
BeyonCa
China

Qiang Liu†
qiangliu@tju.edu.cn
Tianjin University
China

Yinhe Han†
yinhes@ict.ac.cn
ICT, Chinese Academy of Sciences
China

Zishen Wan
zishenwan@gatech.edu
Georgia Tech
USA

Shaoshan Liu
shaoshan.liu@perceptin.io
PerceptIn
USA

Abstract

Despite extensive efforts, existing approaches to design accelerators for optimization-based robotic applications have limitations. Some approaches focus on accelerating general matrix operations, but they fail to fully exploit the specific sparse structure commonly found in many robotic algorithms. On the other hand, certain methods require manual design of dedicated accelerators, resulting in inefficiencies and significant non-recurring engineering (NRE) costs.

The ORIANNA framework represents a significant advancement as it overcomes the limitations of existing approaches and provides a solution to generate accelerators for robotic applications that integrate multiple optimization-based algorithms. The framework leverages a common abstraction factor graph to overcome these challenges. Application designers can easily define their objectives and constraints using the ORIANNA software framework. The ORIANNA compiler then translates the high-level code into basic matrix operations and executes them in an out-of-order manner on the generated accelerator. We demonstrate its effectiveness by implementing a real FPGA-based prototype. ORIANNA accelerators achieve a 6.5× performance improvement and a 15.1× energy reduction compared to an advanced Intel CPU. Furthermore, ORIANNA achieves comparable performance to state-of-the-art dedicated accelerators while utilizing only 36.1% of hardware resources.

*indicates equal contribution to the paper.
†indicates the corresponding author of the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640379

ACM Reference Format:
Yuhui Hao, Yiming Gan, Bo Yu, Qiang Liu, Yinhe Han, Zishen Wan, and Shaoshan Liu. 2024. ORIANNA: An Accelerator Generation Framework for Optimization-based Robotic Applications. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640379

1 Introduction

Autonomous robots have been widely adopted across various domains, providing significant advancements and enhancing human potential. These intelligent machines excel at autonomously performing complex tasks, including industrial automation, healthcare assistance, agricultural operations and domestic chores [1, 50, 51, 59]. Autonomous robots cover a wide range of algorithms and applications, spanning various fields [13]. Among them, optimization-based algorithms play a crucial role and are utilized in multiple tasks such as navigation and localization [52], mission and motion planning [37], decision making and actuation [28].

Optimization-based robotic algorithms involve solving large-scale systems of linear equations iteratively, which introduces considerable computational complexity. For example, a traditional optimization-based localization algorithm [67] can only operate at 5 Hz, even when executed on a state-of-the-art desktop-level CPU [54]. The substantial computation also consumes significant energy, limiting the practicality of utilizing robots.

Architecture communities have proposed two directions to solve the above problems. The first approach is to design programmable optimization solver accelerators [53]. These hardware accelerators directly improve the speed of matrix operations used in optimization-based robotic algorithms, such as matrix multiplication, transpose, and decomposition [28, 37, 52]. Even though they apply to various optimization-based algorithms, the improvements of these accelerators are often limited compared to software-only solutions. This limitation arises from the absence of hardware customization and algorithmic specifications, which are crucial in optimization-based robotic algorithms as they usually involve large yet sparse matrix operations. For instance, in a typical localization problem [10], the coefficient matrix A of the linear equations $Ax = b$ has dimensions of 1126×333, with only 5681 non-zero entries and sparsity of 98.5%.

Another method is to design dedicated hardware accelerators tailored to specific algorithms. It is common to design dedicated accelerators for robotic algorithms by identifying a hardware abstraction (e.g., factor graph [25]) and using it for problem size reduction and hardware customization [19–21, 33, 34, 36, 60–62]. Despite its high-performance capabilities, this approach comes with significant non-recurring engineering (NRE) costs. Even using factor graphs as the same hardware abstraction, different algorithms necessitate completely distinct accelerators due to their unique mathematical formulations and execution pipelines [28, 37, 52]. The variations across algorithms render accelerators non-shareable because none of the computing units can be reused for different algorithms. Stacking different accelerators for a robot application that integrates several algorithms wastes significant chip area.

The limitations of the two approaches stem from the intricate nature of robot kinematics and the diverse array of mathematical formulations found in various algorithms [28, 37, 52]. As depicted in Fig. 1, there is a demand for a methodology that can design a unified hardware accelerator for different robotic algorithms, achieving high performance while minimizing resource consumption and NRE costs.

In this paper, we present ORIANNA, a novel end-to-end framework for generating accelerators specifically designed for robotic applications. Fig. 2 provides an overview of ORIANNA, which utilizes a novel common math representation to build a toolchain, consisting of a factor graph library, an optimizing compiler, and a hardware generation backend. By leveraging factor graph as an abstraction, ORIANNA is capable of generating accelerators for diverse robotic applications (e.g., manipulators, vehicles, drones) containing multiple optimization-based algorithms (e.g., localization, planning).

We comprehensively investigate various mathematical representations for robot pose (orientation and position) utilized in optimization-based algorithms. However, we find that none of these representations are suitable for integrating

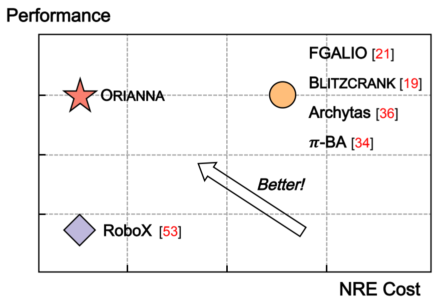


Fig. 1. Comparison of non-recurring engineering (NRE) costs and performance for different works.

the diverse algorithms into a unified hardware accelerator. To address this issue, we propose to utilize a unified form of pose representation denoted as <$\mathfrak{so}(n)$, $\mathbf{T}(n)$>, using a Lie algebra $\mathfrak{so}(n)$ of the special orthogonal group [55] for orientation and a translation vector $\mathbf{T}(n)$ for position. The new pose representation we propose is a standard for the robotic application domain, which can be used in multiple robotic algorithms. We demonstrate that the unified mathematical representation facilitates the seamless transfer of various optimization-based algorithms in robotics to a factor graph inference.

With the unified mathematical representation, we provide a programmable factor graph library for the users to describe their algorithms. The library encompasses a wide range of factors, including various sensor observations and constraints. Users are free to pick any combination of these factors to formulate their optimization-based algorithms in the form of a factor graph inference. Besides, users can customize their own factors based on the ORIANNA factor graph library.

The ORIANNA compiler translates the high-level algorithms in the factor graph formulation into low-level instructions. ORIANNA instruction set architecture (ISA) primarily comprises matrix-related instructions such as matrix multiplication, transpose and decomposition. First, ORIANNA compiler analyzes the high-level program to generate a factor graph and data flow graphs for matrix operations (MO-DFGs). Then, ORIANNA compiler performs a forward traversal and a backward propagation on every MO-DFG within the factor graph to generate instructions for constructing the linear equations. Finally, ORIANNA compiler traverses the factor graph to generate instructions for solving the linear equations.

Based on the instructions generated by the compiler, ORIANNA generates the corresponding hardware accelerators. Although many hardware components, such as matrix multiply units, are generated through templates, ORIANNA automatically generates the data path between different components. At runtime, ORIANNA controller issues the instructions out

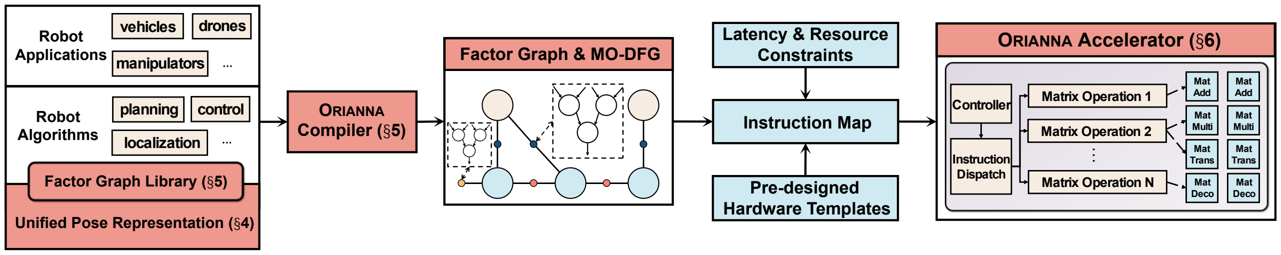


Fig. 2. Overview of ORIANNA. Our proposed unified pose representation facilitates the transformation of various optimization-based robotic algorithms into factor graph inference. Based on the unified pose representation, users can implement diverse robotic algorithms to build their application using ORIANNA factor graph library. Afterward, the ORIANNA compiler translates high-level programs into detailed matrix operations that run on a generated accelerator under user-specific constraints.

of order to burst performance. Our design achieves fine-grained parallelization instead of simply stacking hardware resources.

We demonstrate a prototype on a Xilinx Zynq-7000 SoC ZC706 FPGA[64]. ORIANNA generates accelerators for various real robotic applications, including mobile robots, industrial manipulators, autonomous vehicles and quadrotors. Each application contains multiple optimization-based algorithms. We show that ORIANNA is able to generate accelerators that achieve an average performance improvement of 6.5×, and an average energy reduction of 15.1× compared to an advanced Intel CPU. Compared to a commonly used embedded GPU, ORIANNA achieves 28.6× performance improvement on average and achieves 12.3× energy reduction. When compared to state-of-the-art accelerators, the ORIANNA accelerator achieves comparable performance while consuming only 36.1% of hardware resources. In summary, this paper makes the following contributions:

- We observe that existing accelerators for robotic applications either lack significant performance improvement or incur significant NRE costs and waste tremendous hardware resources. We propose to utilize factor graphs as a common abstraction to generate hardware accelerators for robotic applications automatically that can achieve high efficiency as dedicated accelerators while maintaining low hardware costs.
- We propose a unified mathematical representation for robot pose that is well-suited to formulate multiple optimization-based robotics algorithms into a factor graph inference.
- We design a flexible factor graph software library that allows users to customize their own optimization-based robotics algorithms. We provide implementations for common factors and allow users to extend their customized factors.
- We design a compiler that translates the user program into low-level and detailed instructions. The compiler, along with the software library, saves efforts for hardware designers to map optimization-based robotics programs into a hardware accelerator.
- We are able to generate an accelerator for a complex robotic application and optimize the accelerator under the given hardware resource constraints. We prototype it on a Xilinx Zynq FPGA and show the comparison between state-of-the-art accelerators.

2 Background

We begin by introducing the nonlinear optimization problems in robotic applications (Sec. 2.1) and then present factor graphs, highlighting their connections (Sec. 2.2).

2.1 Nonlinear Optimization in Robotics

Nonlinear optimization refers to the process of finding the optimal values of a set of variables, trying to minimize or maximize a nonlinear objective function [6]. One can often describe it using Equ. 1.

$$x^* = \arg\min_x \|f(x)\|_2^2 \tag{1}$$

The vector x denotes the variables to be optimized, e.g., robot states, and a function $f(x)$ whose result is a vector represents the error function, e.g., errors between ideal sensor models and actual observations. The example objective here is to find the optimal solution x^* that minimizes the 2-norm of the error.

Nonlinear optimization is widely used in all kinds of robotic applications. For instance, in most localization algorithms, the objective is to estimate the precise poses of robots by minimizing the disparity between sensor measurements and a reference model [52]. Planning, on the other hand, entails determining an optimal trajectory considering factors such as path length, smoothness, and obstacle avoidance [37]. In many control algorithms, the goal is to minimize the error between the executed path and the reference path by adjusting the control signals input to the robots [28].

Solving nonlinear optimization problems requires an iterative process of constructing and solving linear equations.

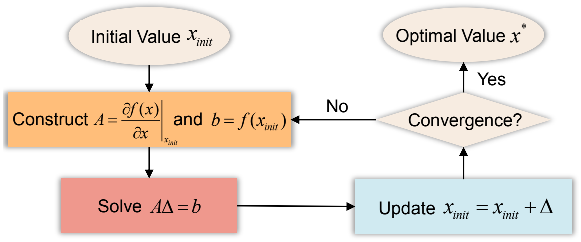

Initial Value x_{init}
Optimal Value x^*
Yes
Construct $A = \frac{\partial f(x)}{\partial x}\big|_{x_{init}}$ and $b = f(x_{init})$
No
Convergence?
Solve $A\Delta = b$
Update $x_{init} = x_{init} + \Delta$

Fig. 3. Gauss-Newton nonlinear optimization process. The optimization starts with an initial value x_{init} and iteratively constructs and solves a series of linear equations $A\Delta = b$. The solution Δ of linear equations is used to update x_{init} until convergence.

Fig. 3 shows widely employed Gauss-Newton approach in solving nonlinear optimization problems [6]. The method first constructs linear equations by calculating the derivatives A of the error function $f(x)$ with respect to the variables x, as well as errors b. Subsequently, it solves the linear equations $A\Delta = b$ which includes applying matrix decomposition techniques such as QR decomposition [14] to formulate an upper-triangular matrix and back substitutions. The result Δ will update the initial point x_{init} until convergence, which occurs when the error becomes smaller than a pre-defined threshold or when the maximum number of iterations is reached. In the end, the optimal solution x^* is obtained.

2.2 Solving Nonlinear Optimization Using Factor Graphs

Factor graph serves as an effective graphical representation employed in large-scale optimization problems in robotics [9]. Specifically, it can be used in solving linear equations $A\Delta = b$ in nonlinear optimization problems. A factor graph is a bipartite graph comprising two types of nodes: variable nodes and factor nodes [25]. Variable nodes represent a set of variables to be optimized and factor nodes represent the constraints among the connected variable nodes.

There exists a direct connection between a factor graph and the coefficient matrix A and the right-hand side (RHS) vector b of the linear equations. Specifically, each factor node aligns with a block row in A and b, and every variable node corresponds to a subset of the solution vector Δ.

Fig. 4 illustrates the correspondence between a factor graph and the linear equations $A\Delta = b$ in localization algorithms. In this example, the variable nodes x_1 to x_3 represent the robot poses, while y_1 and y_2 denote the landmarks. The factor nodes f_1 to f_3 represent measurements from exteroceptive sensors, e.g., camera, while f_4 and f_5 denote measurements from interoceptive sensors, e.g., inertial measurement unit (IMU). Additionally, f_6 represents prior pose information. The arrows between the factor graph and linear equations indicate that factor f_6 corresponds to the last block row of the coefficient matrix A and the RHS vector b.

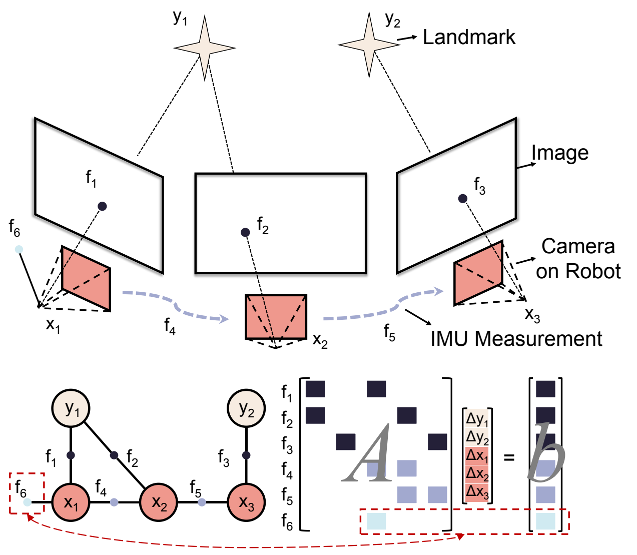


Fig. 4. Correspondence between factor graph and optimization-based robotic localization algorithm. A mobile robot utilizes sensor information from a camera and IMU to estimate its poses and the positions of landmarks in the environment (top). A factor graph can model the localization algorithm (bottom left). There are two types of nodes. x_i and y_i represent variable nodes and f_1 to f_6 denote factor nodes. The factor nodes correspond to the coefficient matrix A and the RHS vector b, while the variable nodes correspond to the solution Δ to linear equations (bottom right).

In most robot algorithms, the coefficient matrix A is of large dimension but exhibits high sparsity [9]. Directly solving such linear equations leads to significant time and resource overhead. However, the interconnections between variables and factors within the factor graph authentically capture the intrinsic structure inherent in the linear equations. Thus, a factor graph inference can help solve the linear equations incrementally, effectively utilizing the underlying sparsity.

The factor graph inference involves a graph traversal along with incremental variable eliminations and back substitutions on the graph [11]. Fig. 5 illustrates the variable elimination process as follows: With a given variable ordering, for each variable, the neighboring factors are first extracted, and the rows in the coefficient matrix A and RHS vector b contained within these factors are concatenated in a row-major order to form a dense matrix $\overline{A}$ with smaller size. Subsequently, a partial QR decomposition is performed on $\overline{A}$, transforming the dense matrix into an upper-triangular one. Afterward, $\overline{A}$ is added back to replace the original rows in A and b. Correspondingly, a new factor (f_7) is added to the graph.

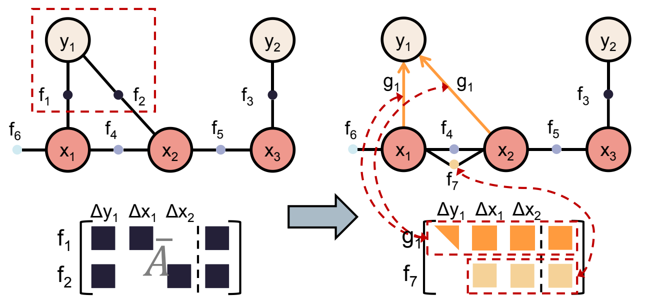


Fig. 5. The variable elimination process during factor graph inference on variable y_1. A small yet dense matrix $\overline{A}$ related to the eliminated factor is formed and partial QR decomposition is performed on it. Afterward, a new factor f_7 is added back to the original graph. New rows and columns from the decomposed matrix will replace the original rows and columns.

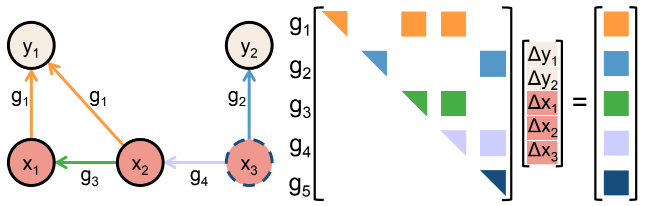


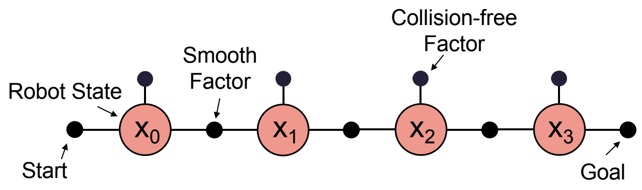
Fig. 6. After eliminating all variables, an updated graph corresponding to an upper triangular matrix is obtained. The solution Δ to the linear equations can be calculated by performing back-substitutions on this graph. Arrows represent the dependency of child variables on the values of their parent variables during the back substitutions. For example, solving x_2 requires the solution of x_3.

Fig. 6 shows that, after performing variable eliminations, the updated graph becomes an upper triangular matrix. Performing back-substitutions on the updated graph starting from the root gets the solution Δ to the linear equations.

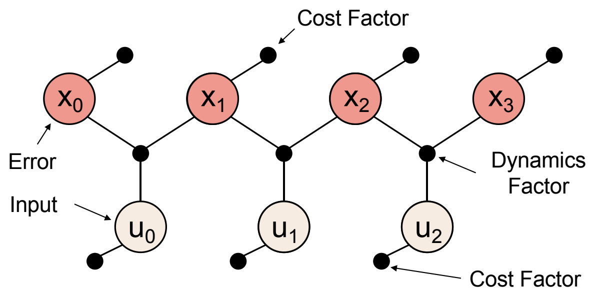
2.3 Factor Graph Examples for Robotic Algorithms

The factor graph provides a powerful abstraction to robotic applications. Besides localization, algorithms in the planning and control domain can also be represented using factor graphs. We show examples of planning and control factor graphs.

Planning factor graph. Motion planning pursues a smooth and collision-free trajectory for robot tasks [19, 31, 40, 71]. For planning factor graphs, variable nodes represent trajectory states, and factor nodes indicate state probability distributions. Factor graph inference aims for optimal solutions maximizing overall probability. Fig. 7a shows the topology for a standard planning algorithm in the form of a factor graph, optimizing for smoothness and collision avoidance. Collision-free factors ensure safe distances with lower probabilities near obstacles, while smooth factors minimize fluctuations with higher probabilities for similar states.



(a) Planning factor graph.



(b) Control factor graph.

Fig. 7. Factor graph examples for different algorithms.

Control factor graph. The control algorithm minimizes the error between reference and actual states of robots by solving optimal control input [20, 28, 38, 58, 65]. Fig. 7b shows the factor graph for the control algorithm. Nodes representing the oldest errors and control signals are on the left, and the newest are on the right. The variable node represents the reference-actual state error. The dynamic factor node models robot dynamics, and the cost factor minimizes error or control input cost. Factor graph inference tries to find input variables achieving zero error in a finite step.

3 ORIANNA Workflow

ORIANNA aims to utilize a factor graph as an abstraction to generate hardware accelerators for optimization-based robotic applications. The goal of ORIANNA is in two folds. First, ORIANNA tries to relieve the effort from software designers by providing unified mathematical representations for robot pose (Sec. 4) along with a factor graph library (Sec. 5.1) for users to design their algorithms and directly transfer them into a factor graph inference. Second, ORIANNA tries to relieve the effort from hardware designers by providing a compiler (Sec. 5.2) to translate high-level programs into low-level instructions. The low-level instructions consist of basic matrix operations. Finally, ORIANNA generates hardware and runs the instructions on the generated accelerators (Sec. 6).

Orianna factor graph library. The workflow for robotic application designers to use Orianna is straightforward. Application designers select the algorithms, such as localization and planning, that they intend to use in the application and program their code using the factor graph library we provide. The process includes adding different variable nodes and factor nodes into an initially empty factor graph.

Orianna compiler. Because of the unified mathematical representations we offer for describing robot pose, Orianna compiler can seamlessly translate the high-level program into low-level instructions by constructing and traversing MO-DFGs (to construct the linear equations) and inferencing factor graphs (to solve the linear equations).

Orianna accelerator. Orianna generates hardware based on the instructions from Orianna compiler and constraints from the application designers. We formulate the hardware generation process as a nonlinear integer programming problem (Sec. 6.2). The hardware generation process generates an accelerator consisting of a set of matrix computing units, such as matrix multiplication and matrix decomposition. During runtime, a controller schedules the instructions in an out-of-order way to boost performance (Sec. 6.3). For example, a matrix multiplication operation inside the localization algorithm and a matrix decomposition operation can happen at the same time as long as there is no data dependency.

4 Unified Pose Representation

This section provides a unified robot pose representation driven by improving compatibility and efficiency (Sec. 4.1). We give the definitions for the unified pose representation (Sec. 4.2) and offer an evaluation to validate its correctness and highlight its benefits (Sec. 4.3).

4.1 Opportunities for Unified Pose Representation

Various forms of pose representation exist in the robot domain. Different algorithms may have different representations even in one robotic application, such as a mobile robot [8]. The localization algorithm may use a combination of a 4-dimensional quaternion $\mathbf{q}$ and 3-dimensional position vector $\mathbf{T}(3)$ [52]. However, the motion planning algorithm can directly use special Euclidean groups $\mathbf{SE}(n)$ and their Lie algebra $\mathfrak{se}(n)$ to represent robot poses [40], where n denotes the dimension. The diverse pose representations result in two limitations of generating an accelerator for robotic applications.

Compatibility. The first limitation is that different math representations for robot poses bring significant challenges to use a common abstraction for different algorithms (e.g., factor graph) and build a general software framework and a compiler to translate different algorithms into a consistent set of instructions. This discrepancy arises due to distinct formulations of optimization objectives resulting from different pose representations, leading to variations in error and derivative formulas. The crucial step in transforming a non-linear optimization solving process into a factor graph inference, i.e., linear equations construction (the yellow block in Fig. 3), will also be different. For example, it is impractical for a localization algorithm using $\mathbf{q}$ and $\mathbf{T}(n)$ and a planning algorithm using $\mathbf{SE}(n)$ and $\mathfrak{se}(n)$ to share the same programming model and compiler when formulating a factor graph inference.

An analogy of the utility of a common pose representation is the IEEE 754 standard. Prior to standardization, different CPU vendors could design their own Arithmetic Logic Units based on their own floating-point representation, and software would be closely bonded with the hardware and not transferable because of a lack of a common representation.

Efficiency. Many representations for robot poses are designed for brevity in mathematical derivation and programmability instead of efficiency. For instance, $\mathbf{SE}(n)$ representation pads zeros and ones under the concatenation of rotation matrix $\mathbf{SO}(n)$ and translation vector $\mathbf{T}(n)$ [4]. Using $\mathbf{SE}(n)$ can express two continuous Euclidean transformations with two variables, while it takes four variables when using a rotation matrix and a translation vector.

However, adding ones and zeros introduces extra computation in constructing and solving the linear equations when optimizing the nonlinear objective. Specifically, when using the $\mathbf{SE}(3)$ representation for a pose in 3-dimensional space, certain operations involve 4-dimensional matrix multiplication and 6-dimensional exponential and logarithmic mapping [4], which leads to more than 2× of extra multiply-accumulate (MAC) operations compared to only using $\mathbf{SO}(n)$ and $\mathbf{T}(n)$.

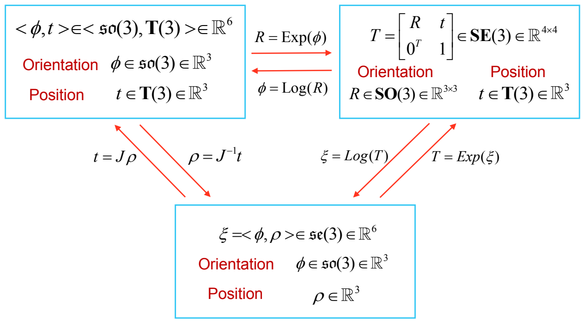
4.2 Unified Pose Representation

We propose to use a unified pose representation $<\mathfrak{so}(n), \mathbf{T}(n)>$ to tackle the above challenges. Specifically, it utilizes $\mathfrak{so}(n)$ to represent the orientation and a vector $\mathbf{T}(n)$ to represent the position. $\mathfrak{so}(n)$ is the vector form of the rotation matrix $\mathbf{SO}(n)$, which can be transformed into $\mathbf{SO}(n)$ using the exponential mapping [4]. Equ. 2 defines addition and subtraction operations for this representation:

$$\begin{aligned} \xi_1 \oplus \xi_2 &= <\mathrm{Log}(\mathrm{R}_1\mathrm{R}_2), \quad t_1 + \mathrm{R}_1 t_2> \\ \xi_1 \ominus \xi_2 &= <\mathrm{Log}(\mathrm{R}_2^{\mathrm{T}}\mathrm{R}_1), \quad \mathrm{R}_2^{\mathrm{T}}(t_1 - t_2)> \\ \mathrm{R}_1 &= \mathrm{Exp}(\phi_1), \quad \mathrm{R}_2 = \mathrm{Exp}(\phi_2) \end{aligned} \tag{2}$$

where ξ_1, ξ_2 belong to $<\mathfrak{so}(n), \mathbf{T}(n)>$, ϕ_1, ϕ_2 belong to $\mathfrak{so}(n)$, R_1, R_2 belong to $\mathbf{SO}(n)$, and t_1, t_2 belong to $\mathbf{T}(n)$. $\mathrm{Exp}(\cdot)$ and $\mathrm{Log}(\cdot)$ are exponential and logarithmic mapping, respectively, enabling transformation between $\mathbf{SO}(n)$ and $\mathfrak{so}(n)$ [55].

By applying the unified pose representation $<\mathfrak{so}(n), \mathbf{T}(n)>$ and treating operations $\oplus$, $\ominus$ as primitive operations, we can represent various complex robot kinematics using the



$< \phi, t > \in < \mathfrak{so}(3), \mathbf{T}(3) > \in \mathbb{R}^6$
Orientation $\phi \in \mathfrak{so}(3) \in \mathbb{R}^3$
Position $t \in \mathbf{T}(3) \in \mathbb{R}^3$
$R = \mathrm{Exp}(\phi)$
$\phi = \mathrm{Log}(R)$
$T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \in \mathbf{SE}(3) \in \mathbb{R}^{4\times4}$
Orientation $R \in \mathbf{SO}(3) \in \mathbb{R}^{3\times3}$
Position $t \in \mathbf{T}(3) \in \mathbb{R}^3$
$t = J\rho$
$\rho = J^{-1}t$
$\xi = Log(T)$
$T = Exp(\xi)$
$\xi = < \phi, \rho > \in \mathfrak{se}(3) \in \mathbb{R}^6$
Orientation $\phi \in \mathfrak{so}(3) \in \mathbb{R}^3$
Position $\rho \in \mathbb{R}^3$

Fig. 8. The transformation among three pose representations, which include our proposed representation $<\mathfrak{so}(3), \mathbf{T}(3)>$ (top left), Special Euclidean Group $\mathbf{SE}(3)$ (top right), and the Lie algebra of $\mathbf{SE}(3)$ denoted as $\mathfrak{se}(3)$ (bottom).

composition of these primitive operations. For example, localization and control algorithms commonly employ $\ominus$ to calculate the deviation between actual observations and ideal models [28, 54], while planning algorithms utilize $\oplus$ to compute the world coordinates of points on robot links [40].

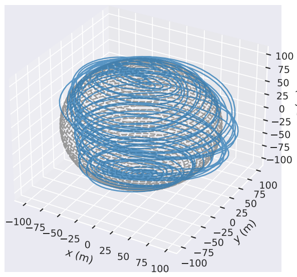
4.3 Unified Pose Representation Validation

It has been proved that various pose representations can be converted interchangeably [55]. We first show how different pose representations are equivalent using two common pose representations. We then validate that using $<\mathfrak{so}(n), \mathbf{T}(n)>$ pose representation does not sacrifice any accuracy in a localization example [16].

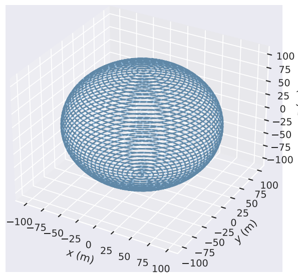
We show the equivalence among two common pose representations and ours in Fig. 8. The transformation between $<\mathfrak{so}(3), \mathbf{T}(3)>$ and $\mathbf{SE}(3)$ is straightforward, involving the exponential and logarithmic mapping between $\mathfrak{so}(3)$ and $\mathbf{SO}(3)$, which mainly consists of matrix multiplication, addition and matrix-vector multiplication. The transformation between $\mathbf{SE}(3)$ and $\mathfrak{se}(3)$ also involves exponential and logarithmic mapping in higher dimensions, thus the computation is more complex compared to the transformation between $\mathfrak{so}(3)$ and $\mathbf{SO}(3)$. The transformation between $<\mathfrak{so}(3), \mathbf{T}(3)>$ and $\mathfrak{se}(3)$ requires a linear mapping $J \in \mathbb{R}^{3\times3}$ over the position component [55].

Besides showing how our new pose representation is equivalent to existing representations, we also use a complex benchmark that involves localization in a 3-dimensional scenario for validation. The ground-truth trajectory forms a sphere composed of multiple layers ascending from bottom to top. Each layer should form a perfect circle. In reality, the perfect circle will not be formed if we do not perform optimization because of different kinds of sensor noise, as we show in Fig. 9a.

We optimize the noisy trajectory using the $<\mathfrak{so}(3), \mathbf{T}(3)>$ pose representation and show the results in Fig. 9b. Tbl. 1 presents the absolute trajectory errors. The average trajectory error compared to the ground truth is a mere 7 mm. Compared to the trajectory optimized using traditional $\mathbf{SE}(3)$, our representation does not lose accuracy.



(a) Initial trajectory obtained from a sensor with noise (the blue line).



(b) Trajectory optimized using the pose representation we propose (the blue line).

Fig. 9. Trajectory accuracy comparison. Gray dashed lines represent ground truth.

Table 1. Absolute trajectory errors (unit: meters) of initial trajectory and pose-optimized trajectories using $<\mathfrak{so}(3), \mathbf{T}(3)>$ and $\mathbf{SE}(3)$ pose representation. **Max** denotes the maximum trajectory error. **Mean** represents the average trajectory error. **Min** shows the minimum trajectory error. **Std** indicates the standard deviation.

	Max	**Mean**	**Min**	**Std**
Initial Error	62.695	17.671	0.595	9.998
$<\mathfrak{so}(3), \mathbf{T}(3)>$	0.036	0.007	0.000	0.005
$\mathbf{SE}(3)$	0.037	0.007	0.000	0.005

Moreover, employing $<\mathfrak{so}(3), \mathbf{T}(3)>$ representation saves significant overhead compared to existing approaches, particularly when compared to $\mathbf{SE}(3)$, as it avoids padding zeros and ones, resulting in matrix operations with smaller size. Specifically, we statistically save 52.7% of MAC operations. We will use $<\mathfrak{so}(n), \mathbf{T}(n)>$ to represent robot poses in all later sections.

5 Orianna Software Framework

We offer intuitive software frameworks to application designers, allowing them to program different algorithms into a factor graph inference easily. Traditional software framework such as GTSAM [27] necessitates users to possess a high level of expertise in the robotic domain. For example, to implement a localization algorithm using GTSAM, the users need to manually program the pose representation, the sensor model, cost functions and analytical derivatives.

The goal of our software framework is twofold. First, we aim to eliminate the need for software designers to manually calculate the coefficient matrix and RHS vector, as this

Table 2. Type of factors supported in the factor graph library.

Factor Type	Factor	Algorithm
Measurement	LiDAR, Camera, GPS, IMU, Prior	Localization [52, 54]
Constraint	Smooth, Collision-free, Kinematics, Dynamics	Planning [40], Control [65]

process can be complex and varies among different algorithms. Second, we seek to simplify the formulation of linear equations solving into a factor graph inference. Finally, we endeavor to automate the translation of high-level implementations into detailed matrix operations to facilitate program execution on an accelerator.

To achieve this, we completely change the programming model for optimization-based robotic applications. We first provide a factor graph library (Sec. 5.1) where users can build their applications by choosing different factors. Users are also granted the flexibility to customize their own factors. We abstract away all the mathematical details from the users, such as constructing and solving linear equations. Instead, these steps are seamlessly handled by ORIANNA compiler (Sec. 5.2).

5.1 Factor Graph Library and Programming Model

ORIANNA provides users with a powerful and flexible factor graph library. We summarize the type of factor nodes we support in Tbl. 2. The library includes a variety of factor nodes organized into two categories. The first category includes measurement factors derived from sensor observations such as LiDAR measurements. The second class consists of various constraint factors. For example, the kinematics constraint factor can be utilized in both planning and control algorithms to constrain the maximum speed limit of the robot.

Underlying these factors are different types of matrices and vectors, whose complexity is hidden from the users. For instance, a factor node represents camera observation (e.g., f_1 in Fig. 4) corresponding to two matrix blocks with dimensions of two rows and six columns and two rows and three columns, respectively, along with one vector of length two.

Programming model. Based on the factor graph library we provide, the programming model for the users is completely changed. The users can program their robotic application by gradually constructing a factor graph. To illustrate this paradigm, we present an example of implementing a localization algorithm into a factor graph [52]. Assuming the sliding window has three keyframes and two landmarks, as we show in Fig. 4.

```
#Localization graph
graph.add(CameraFactor(x1, y1, m1))
graph.add(CameraFactor(x2, y1, m2))
graph.add(CameraFactor(x3, y2, m3))
graph.add(IMUFactor(x1, x2, m4))
graph.add(IMUFactor(x2, x3, m5))
graph.add(PriorFactor(x1, p1))
graph.optimize()
```

Users start with an empty factor graph and gradually add factor nodes and variable nodes to the graph. In this case, the robot poses x are represented using <$\mathfrak{so}(n), \mathbf{T}(n)$> representation, and variables y indicate landmarks. The constants m denote measurements from sensors.

The dimensions of variables x and y are determined by the specific application. For instance, in the case of a drone, the poses x are six-dimensional, where three dimensions denote orientation and three dimensions represent position. The landmarks y are three-dimensional, indicating position. A camera measurement factor `CameraFactor` is added between y1 and x1 as the drone observes y1 at pose x1. The variable m1 represents the observation of landmark y1 when the robot is at pose x1, forming the pixel coordinates captured in factor f1 in Fig. 5. IMU factors `IMUFactor` are added between adjacent pose variables (between x1 and x2, x2 and x3) to integrate the acceleration and gyroscope measurements. m4,m5 represent IMU observations. Finally, a prior factor `PriorFactor` is added between x1 and previous pose p1 to fix the absolute pose of the drone.

Customized factors. Users can extend customized factors by providing the error expression ($f(x)$ in Equ. 1) to ORIANNA. For instance, if users want to incorporate a new factor that is a constraints type to their applications, enforcing constraints between two robot poses, they only need to provide the error function as:

$$f(x_i, x_j) = (x_i \ominus x_j) \ominus z_{ij} \tag{3}$$

where constant z_{ij} represents the constraint, and x_i and x_j are robot poses to be optimized. The symbol $\ominus$ is the subtraction in our proposed pose representation, denoting the difference between two poses. The ORIANNA compiler automatically generates instructions for computing errors and derivatives by analyzing the user-provided new factor code (Sec. 5.2).

5.2 ORIANNA Compiler

The ORIANNA compiler manages two data structures: a factor graph and a matrix operation data flow graph (MO-DFG) for each factor node within the factor graph. Instructions are then generated based on these data structures. The ORIANNA compiler follows the subsequent steps in its process. First, ORIANNA compiler will go through the user code to generate a factor graph. Second, ORIANNA compiler analyzes the high-level program to generate MO-DFGs for all factor nodes in the factor graph. Next, it traverses the MO-DFGs forwardly to generate instructions for constructing the RHS vector b of the linear equations, and it performs backward propagation on MO-DFGs to generate instructions for constructing the

Table 3. Primitive operation types.

Operation	Description
VP	Vector addition (subtraction)
RT	Rotation matrix transpose
Log	Logarithmic mapping of rotation matrix [12]
RR	Rotation matrix multiplication
RV	Rotation matrix-vector multiplication
Exp	Exponential mapping of Lie algebra [12]
$(\cdot)^\wedge$	Skew-symmetric matrix
$\mathrm{J_r}(\cdot)$	Right Jacobian [55]
$\mathrm{J_r^{-1}}(\cdot)$	Right Jacobian inverse [55]

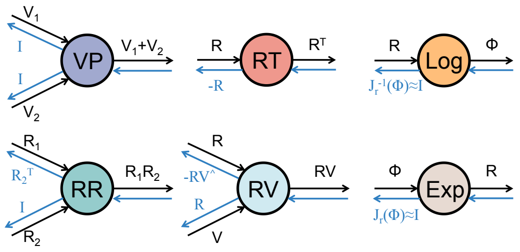


Fig. 10. Primitive operations involved in robot kinematics using our unified pose representation. The forward arrows (black) depict the operation process, while the backward arrows (blue) represent the differentiation process.

coefficient matrix A, which is feasible due to the chain rule of differentiation [7]. Finally, Orianna compiler traverses the factor graph to generate instructions for factor graph inference.

MO-DFG. Based on the unified math representations we propose in Sec. 4, we only have nine different primitive matrix operations[1] during forward traversal and backward propagation on the MO-DFGs, as we show them in Tbl. 3.

Fig. 10 shows that each primitive matrix operation corresponds to a node in the MO-DFG[2], where the forward arrows of a node represent an operation on the operands, and the reverse arrows indicate a derivative with respect to the operands. For instance, the primitive operation of rotation matrix multiplication is present in both addition and subtraction operations in Equ. 2, corresponding to the RR node. Forward arrows indicate the matrix multiplication computation between R_1 and R_2. Backward arrows represent the derivatives of the result with respect to R_1 and R_2, yielding the transpose of R_2 and the identity matrix I [55], respectively.

We use the example in Equ. 3 to illustrate the workflow of Orianna compiler on MO-DFGs. Orianna compiler generates the MO-DFG through two steps. First, by leveraging Equ. 2, Orianna compiler transforms Equ. 3 to Equ. 4, which

[1]Regular matrix-vector multiplication can reuse the RV primitive.

[2]Primitive operations in Fig. 10 correspond to $<\mathfrak{so}(3), \mathbf{T}(3)>$ in three-dimensional space. In two-dimensional space, the primitive operations are the same, except for slight differences in the results of back propagation [55].

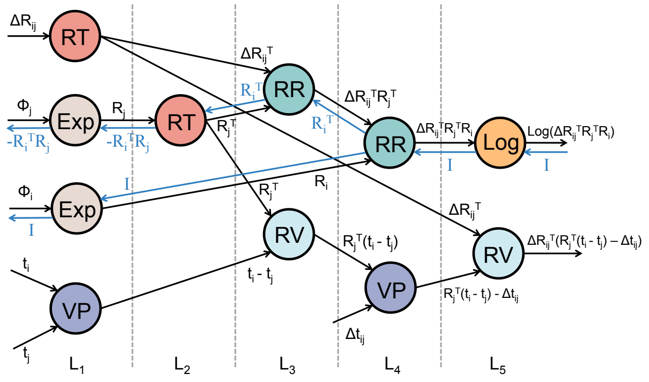


Fig. 11. MO-DFG generated for Equ. 4. A series of primitive operations constitute a complex calculation. Forward traversal on the MO-DFG yields the error instructions. Backward propagation on the MO-DFG produces the derivative instructions. For the sake of brevity, here is the entire error calculation process, as well as a part of the derivative calculation process. Instructions within the same layer can be executed in parallel since no data dependencies exist between them.

gives the error expressions for the orientation and position components consisting of a series of primitive operations:

$$\begin{aligned} e_o &= \text{Log}(\Delta R_{ij}^T R_j^T R_i) \\ e_p &= \Delta R_{ij}^T (R_j^T (t_i - t_j) - \Delta t_{ij}) \end{aligned} \tag{4}$$

where ΔR_{ij}, R_i and R_j (Δt_{ij}, t_i and t_j) represent the rotation matrices (translation vectors) of constraint and two poses, respectively. Second, Orianna compiler goes through Equ. 4 to build a MO-DFG. Specifically, Orianna compiler will generate the postfix expressions of Equ. 4 and parse the postfix expressions using a stack data structure to get the MO-DFG.

Fig. 11 illustrates the generated MO-DFG for Equ. 4. The MO-DFG will be used to create coefficient matrix A and RHS vector b. To generate b, Orianna compiler performs a forward traversal (from left to right on Fig. 11) on the MO-DFG to yield instructions for computing the error. To generate A, Orianna performs backward propagation (from right to left on Fig. 11) on the MO-DFG to produce instructions for computing the derivatives. For instance, we can obtain the derivative $\partial e_o / \partial \phi_j$ of the orientation error with respect to the orientation variable of pose j as $-R_i^T R_j$. Due to the data dependencies, we adopt the breadth-first search (BFS) to traverse the MO-DFG in both forward and backward processes, enabling the maximum parallel execution of the instructions.

Factor graph. Every factor node in the factor graph has its own MO-DFG. After the instructions from each MO-DFG are generated, Orianna compiler traverses the entire factor graph in a given variable order to get the entire instruction set for the matrix decomposition step (e.g., Fig. 5). This is

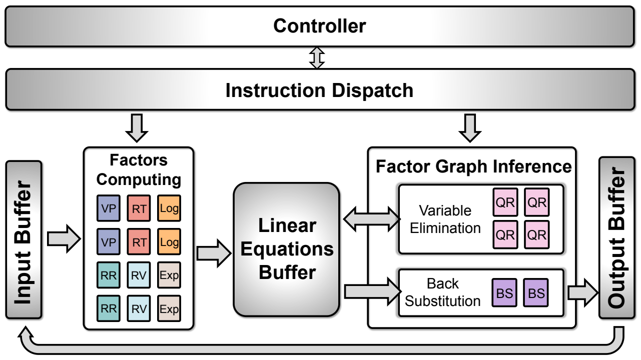


Fig. 12. Overall hardware architecture.

the last step of Orianna compiler. Both instructions for performing QR decomposition and back substitution are generated at this stage. The instructions of QR decomposition are generated through the given order of variable nodes as we illustrated in Fig. 5. As the factor graph is constantly updated every time one variable is eliminated, after the graph is updated, Orianna compiler will traverse the new graph with the same step to generate a new set of instructions to eliminate the next variable. Finally, when the matrix A becomes an upper-triangular matrix, the graph is fixed. We generate the instructions of back substitution through the reverse order as Fig. 6 at last.

6 Hardware Generation

This section presents the Orianna hardware. We first introduce the basic Orianna hardware architecture (Sec. 6.1). Although Orianna generates the accelerator through different basic hardware templates, the Orianna software framework allows to optimize the accelerator in two forms. First, as we have accurate latency and energy estimations of different basic matrix operations, we are able to customize the hardware based on different user constraints (Sec. 6.2). Second, we can provide both coarse-grained out-of-order execution between different algorithms and fine-grained out-of-order execution inside every algorithm (Sec. 6.3).

Notice that although the hardware generation relies on pre-defined templates that we will describe in Sec. 6.1, the novelty of our generation process can be summarized in two folds. First, the connections between different circuit blocks are automatically generated based on the dedicated data flow of the matrix operations. Second, Orianna can find optimal hardware configurations based on given resource constraints. For example, given the hardware resource constraints, Orianna could optimally determine how much should be used to build a matrix decomposition unit and how much should be used on a matrix multiplication unit.

6.1 Template-based Hardware Generation

Orianna generates an accelerator capable of processing an entire robotic application containing multiple algorithms. We achieve it by relying on the factor graph inference as a unified abstraction. The hardware is designed to implement different basic matrix operations, such as matrix multiplication and decomposition. Thus, these basic blocks are generated through different templates. For example, for matrix multiplication, we apply widely used systolic array architecture as our template. The matrix decomposition unit we use is basic QR decomposition, which is similar to prior works [19, 21, 36].

Fig. 12 illustrates the Orianna hardware architecture. It consists of two parts, computing factors to construct linear equations and solving them. We name them factor computing block and factor graph inference block. The first part supports all the primitive matrix operations in Tbl. 3 and the second part supports matrix decomposition and back substitution.

The factors computing block starts with the current variables x_{init} (along with observations) as input to calculate the error and derivative terms involved in the objective function by traversing the MO-DFGs (e.g., Fig. 11). The error and derivative terms are stored in an on-chip buffer. The factor graph inference block solves the linear equations utilizing incremental QR decompositions and back substitutions. Each step of this process involves solving a small and dense matrix, taking advantage of the sparse structures in the linear equations. After each iteration, the solution Δ is used to update x_{init} to check whether the convergence requirements are satisfied.

The simple hardware architecture can accommodate a wide range of optimization-based robotic algorithms utilizing the unified pose representation and factor graph as an abstraction. The only variation occurs in the factor computing block, where distinct algorithms may yield different primitive matrix operations after compilation.

6.2 Constraints-based Hardware Optimization

Orianna accelerators execute instructions which are basic matrix operations generated by Orianna compilers. As we have accurate estimations of the latency and energy consumption of each instruction, we are able to optimize the hardware to meet goals based on different kinds of constraints. Users can specify constraints such as available hardware resources. They can also specify their goals such as achieving lower average frame latency and solving long tail problems by achieving lower maximum frame latency and lower energy consumption.

We use an example of optimizing average frame latency under limited hardware resources to illustrate the process. In this case, the Orianna hardware generation process can be formulated as a nonlinear integer programming problem:

$$\begin{aligned} p_1^*, p_2^*, \cdots, p_n^* &= \underset{p_1, p_2, \cdots, p_n}{\arg\min}\; L(p_1, p_2, ..., p_n) \\ s.t. \quad & R(p_1, p_2, ..., p_n) \le \mathrm{R}^*, \end{aligned} \tag{5}$$

$L(\cdot)$ represents the computation latency. $R(\cdot)$ indicates the hardware resource consumption, and R^* denotes the maximum on-chip resources. $p_1, ..., p_n$ signifies the number of replicated computation units employed for various matrix operations.

At first, only one computation unit is instantiated for each matrix operation block. To optimize the average frame latency, ORIANNA identifies the critical path in MO-DFGs and factor graphs, gradually increasing the number of computation units by solving Equ. 5. After adding one computation unit hardware, ORIANNA will check two things. First, it will check whether the hardware resources limitation has been violated to determine whether the process should continue. Second, it will check whether the critical path changes to determine how to solve Equ. 5 in the next step.

6.3 Out-of-Order Execution

One of the key advantages of using factor graph as a unified abstraction for diverse robotic algorithms is that ORIANNA is able to translate different algorithms into the same set of matrix operations and allow both fine-grained out-of-order execution and coarse-grained out-of-order execution. The out-of-order instruction dispatching happens at both the linear equations construction phase and the solving phase.

Fine-grained out-of-order execution inside one algorithm. The out-of-order execution inside one algorithm can occur at two places. First, instructions within the same MO-DFG can be executed out-of-order as long as there is no data dependency (e.g., RR and RV primitive operations in level L_3 in Fig. 11). Second, instructions belonging to different MO-DFGs can be executed in an out-of-order way.

Coarse-grained out-of-order execution between different algorithms. ORIANNA enables out-of-order execution of instructions between different algorithms within a robotic application. For example, instructions from localization and planning algorithms, devoid of data dependencies, can be executed out-of-order.

Considering that algorithms within a single robotic application often have varying execution frequencies, such as the planning algorithm exhibiting a much lower frequency than the localization and control algorithms in an industrial manipulator [39], the coarse-grained out-of-order execution allows ORIANNA accelerators to achieve comparable performance compared to state-of-the-art accelerators but with significantly reduced hardware resources, as the ORIANNA hardware is always fully pipelined.

Out-of-order in variable elimination and back substitution. After the linear equations are constructed, the solving part also follows an out-of-order manner. During factor graph inference, ORIANNA employs a forward-looking approach in the variable order. In variable eliminations, if consecutive variables do not have common adjacent factors, it indicates that there is no data dependency between the elimination of these variables, allowing for out-of-order elimination. (e.g., variables y_1 and y_2 in Fig. 5). In back substitutions, if consecutive variables share the same parent node, it denotes that there is no data dependency between the back substitutions of these variables, enabling out-of-order back substitutions. (e.g., variables x_2 and y_2 in Fig. 6).

Table 4. Evaluation benchmark and node information (dimensions of variable nodes and types of factor nodes) in factor graphs for different algorithms. The two numbers in the control algorithm denote the dimensions of the state and input variables, respectively.

		Localization	Planning	Control
MOBILEROBOT	Variable dim	3	6	3, 2
	Factor	LiDAR GPS	Collision-free Smooth	Dynamics
MANIPULATOR	Variable dim	2	4	2, 2
	Factor	Prior	Collision-free Smooth	Dynamics
AUTOVEHICLE	Variable dim	3	6	5, 2
	Factor	LiDAR GPS	Collision-free Kinematics	Kinematics Dynamics
QUADROTOR	Variable dim	6	12	12, 5
	Factor	Camera IMU	Collision-free Kinematics	Kinematics Dynamics

7 Evaluation

This section starts with a description of our experimental setup (Sec. 7.1). Next, we present the accuracy results of ORIANNA compared to the software implementation (Sec. 7.2). We then evaluate the performance improvement and energy reduction achieved by ORIANNA compared to advanced CPUs (Sec. 7.3). Additionally, we assess the performance and energy gains achieved by ORIANNA when compared to state-of-the-art accelerators (Sec. 7.4). Finally, we conduct a hardware optimization analysis of ORIANNA (Sec. 7.6).

7.1 Evaluate Setup

Benchmark. We evaluate ORIANNA accelerator on four robotic applications. Tbl. 4 describes the different applications and their nodes in factor graphs for various algorithms. Specifically, MOBILEROBOT refers to a two-wheeled robot moving on a two-dimensional plane [26]. MANIPULATOR represents a two-link robot arm [41]. AUTOVEHICLE denotes a four-wheeled unmanned vehicle with car dynamics constraints [22]. QUADROTOR is a four-rotor micro drone [2].

For example, MOBILEROBOT has all three algorithms in the pipeline. In the localization algorithm, the variable nodes represent three-dimensional poses, comprising two-dimensional position and one-dimensional orientation. LiDAR factors and

GPS factors are used to estimate robot pose. For the planning algorithm, the variable nodes contain the robot pose (x, y, θ) and velocities $(\dot{x}, \dot{y}, \dot{\theta})$, contributing to a total of six dimensions. Collision-free factors and smooth factors are used to determine an optimal path. The variable nodes in the control algorithm consist of two parts: a three-dimensional pose and two-dimensional linear and angular velocities. Dynamics factors are used to adjust control values.

Hardware setup. We synthesize the accelerator using Vivado 2021.1 and implement it on the Xilinx Zynq-7000 SoC ZC706 FPGA [64]. The accelerator operates at a fixed frequency of 167 MHz. We employ the Vivado power analysis tool to estimate power consumption. All power and resource utilization data are obtained after the successful completion of the post-layout timing analysis.

Software setup. We implement all four benchmark applications using our proposed pose representation (Sec. 4) and factor graph library (Sec. 5). We compare the results between our implementation and a software implementation using GTSAM [27] to get the accuracy metrics.

Baselines. We compare ORIANNA with six different baselines, which represent the state-of-the-art hardware solutions to current optimization-based robotic applications.

- High-end desktop CPUs. We compare ORIANNA with a high-performance 16-core Intel 11th generation i7-11700 CPU that operates at 2.5GHz. We refer to Intel in this section.
- High-end desktop CPUs with our new pose representation. We create a new software baseline using our pose representation. We aim to demonstrate that although our pose representation saves parameters and computation in theory, the software-only solution does not provide enough performance improvement without the co-design of ORIANNA hardware. We refer this baseline to ORIANNA-SW.
- Lower power mobile CPUs. We compare ORIANNA with a quad-core Arm Cortex-A57 processor on the NVIDIA mobile Jetson TX1 platform [49] operates at 1.9GHz. We refer to ARM in this section.
- Embedded GPUs. We accelerate various robotic applications by leveraging an embedded NVIDIA Maxwell GPU [49]. We implement the GPU baseline using the cuBLAS library [47] to perform matrix operations during linear equations construction and the cuSolverSP library [48] which is built based on cuBLAS and cuSPARSE on matrix decomposition and back substitution operations to solve the linear equations in a sparse manner. We refer this baseline to GPU.
- Accelerator for dense matrix operations. We try our best to implement a hardware that directly accelerates matrix operations used in optimization problems. This baseline is implemented using state-of-the-art high level synthesis (HLS) techniques with manual modifications. This baseline shares all the basic matrix operation units with ORIANNA, such as using a systolic array for matrix multiplication and using the same QR decomposition template. The accelerator is programmable but does not utilize a factor graph as an abstract, thus it does not apply the sparse structure we find. We refer to VANILLA-HLS in this section.
- Accelerator utilizes factor graphs to accelerate individual algorithms. We try our best to implement three existing accelerators that utilize factor graphs to accelerate different algorithms for localization [21], planning [19] and control [20]. As they do not share anything in common in pose representation, software framework and hardware architecture, we stack them together to compare with ORIANNA. We refer them to STACK in this section.

Table 5. Success rate comparison between ORIANNA and software implementation with [27].

Application	MOBILEROBOT	MANIPULATOR	AUTOVEHICLE	QUADROTOR
Software with [27]	100%	96.7%	100%	93.3%
ORIANNA	100%	96.7%	100%	93.3%

ORIANNA Variants. We provide two variants of ORIANNA. ORIANNA-IO executes instructions in an in-order way and ORIANNA-OoO performs out-of-order executions.

7.2 Accuracy

We first show the accuracy comparison between ORIANNA and the software implementation using [27] in Tbl. 5. As each application contains more than one algorithm, we use the mission success rate as the metric here. For instance, in MOBILEROBOT, a mission is considered successful if it can navigate from the starting point to the destination location within the specified time and along the planned path.

It can be seen that with the unified pose representation and new programming model, ORIANNA achieves the same success rate compared to the software using [27].

7.3 Performance and Energy Consumption

We compare the average latency to show the improvement ORIANNA provides on performance. Fig. 13 illustrates the speedup of ORIANNA compared to other baselines. We normalize the latency to ARM. Accelerating robotic applications with an embedded GPU does not improve the performance significantly. The performance of GPU is only 2.03× compared to ARM. We notice that although the performance of the linear equations construction process can be largely improved (up to 4.8×), the total performance improvement of using GPU is not significant, even when we treat all the matrices as sparse ones and accelerate the baseline using libraries designed for sparse matrix operations. We think the fundamental reason is that the linear equations construction

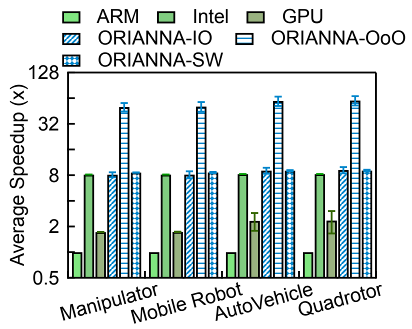


Fig. 13. Speedup comparison between ORIANNA-OoO, ORIANNA-IO, Intel and ARM. Normalized to ARM. The average speedup brought by ORIANNA-OoO is 53.5× across different applications, with a standard deviation of 5.3×.

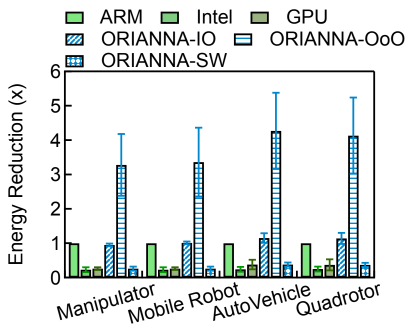


Fig. 14. Energy consumption comparison between ORIANNA-OoO, ORIANNA-IO, Intel and ARM. Normalized to ARM. The average energy reduction brought by ORIANNA-OoO is 3.4× across different applications, with a standard deviation of 0.5×.

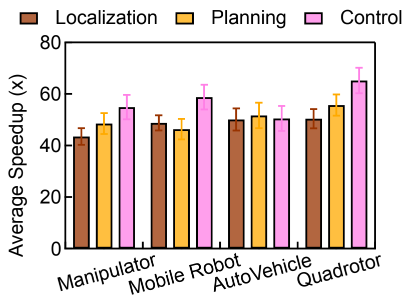
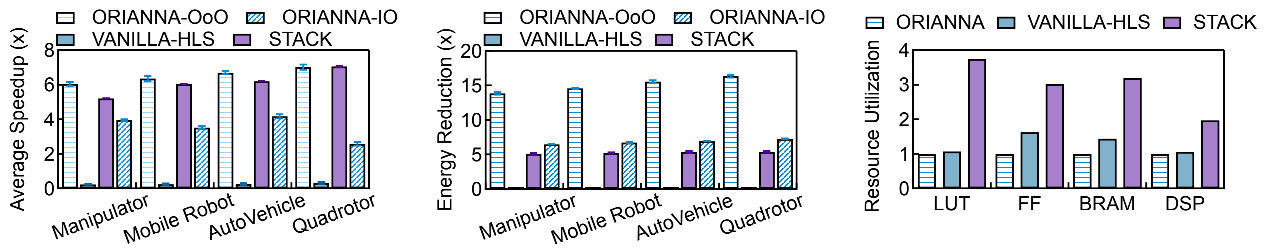


Fig. 15. Speedup breakdown in different algorithms over ARM. Normalized to ARM. On average, localization has a speedup of 48.2× with a standard deviation of 3.2×. The average speedup and standard deviation for planning are 50.6× and 4.1×. The numbers are 60.7× and 4.7× for control.



(a) Speedup comparison with Intel between ORIANNA and other accelerators. Normalized to Intel. The average speedup brought by ORIANNA-OoO is 6.5× across different applications, with a standard deviation of 0.4×.

(b) Energy consumption comparison with Intel between ORIANNA and other accelerators. Normalized to Intel. The average energy reduction brought by ORIANNA-OoO is 15.1× across different applications, with a standard deviation of 1.1×.

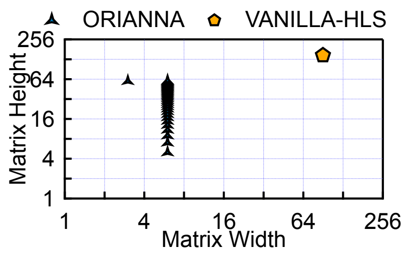
(c) Resource consumption comparison. ORIANNA-OoO and ORIANNA-IO has same resource consumption. On average, ORIANNA-OoO saves 20% of the hardware resource compared to VANILLA-HLS and 64% for STACK.

Fig. 16. Comparison between ORIANNA and state-of-the-art accelerators.

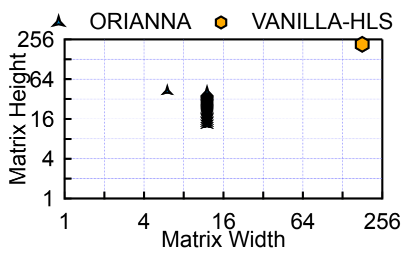
process only takes up to 16% of the total latency and for the majority of the latency that is spent on matrix decomposition and back substitution, the speed up is not significant due to the reason that the sparsity in the matrix operation is non-structural. ORIANNA-SW shows that applying our pose representation in software only results in less than 10% performance improvement, which means the potential of new pose representation itself can not be fully utilized in software, making hardware acceleration necessary. On average, ORIANNA-OoO demonstrates a significant speedup of 53.5× over ARM, 6.5× over Intel and 28.6× over GPU. Executing instructions out-of-order has a significant impact on performance. ORIANNA-OoO has an average 6.3× speedup compared to ORIANNA-IO.

Fig. 14 presents the energy reduction achieved by the ORIANNA accelerator. ORIANNA-OoO achieves an average energy reduction of 15.1× over the Intel, 3.4× over the ARM and 12.3× over the GPU. Compared to ORIANNA-IO, the out-of-order design ORIANNA-OoO has a 2.2× energy reduction. This can be attributed to the fact that ORIANNA-OoO allows more data to be stored on-chip and reused, saving energy on data movement.

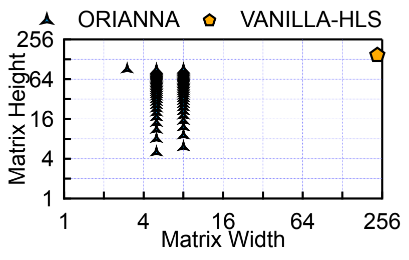
Fig. 15 shows the breakdown of speedup in each algorithm across various applications. Compared to the ARM, the ORIANNA-OoO achieves an average speedup of 48.2×, 50.6×, and 60.7× in localization, planning, and control algorithms, respectively. Particularly, the speedup in control algorithms stands out due to the optimization problems in these algorithms typically having the highest dimensions of optimization variables, enabling more aggressive parallel execution of instructions.



(a) Matrix operation size comparison between VANILLA-HLS and Orianna in localization in mobile robot application.

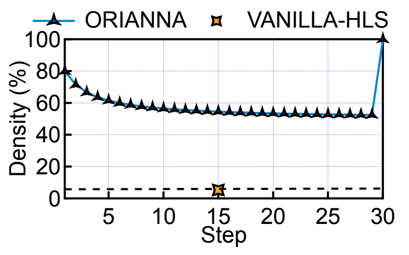


(b) Matrix operation size comparison between VANILLA-HLS and Orianna in planning in mobile robot application.

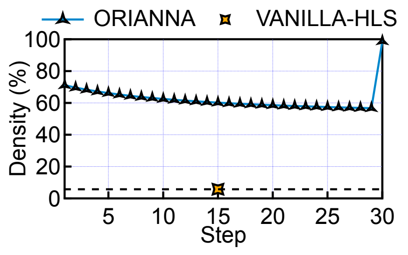


(c) Matrix operation size comparison between VANILLA-HLS and Orianna in control in mobile robot application.

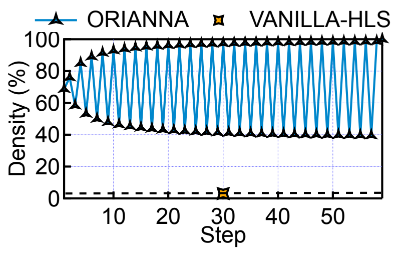
Fig. 17. Comparison between Orianna and VANILLA-HLS on the size of matrix operations. Lower is better.



(a) Matrix operation density comparison between VANILLA-HLS and Orianna in localization in mobile robot application.



(b) Matrix operation density comparison between VANILLA-HLS and Orianna in planning in mobile robot application.



(c) Matrix operation density comparison between VANILLA-HLS and Orianna in control in mobile robot application.

Fig. 18. Comparison between Orianna and VANILLA-HLS on the density of matrix operations. Higher is better.

We further show the breakdown of the latency in each matrix operation in Orianna accelerators. In the example of the drone application, the most time-consuming part is matrix decomposition, which takes 74.0% of the total latency. Constructing matrix A and RHS vector b with the matrix operations we show in Tbl. 3 takes 16.0% of the total latency and back substitution consumes the last 10.0% of the latency.

7.4 Comparison between Orianna and State-of-the-art Accelerators

We also compare Orianna with existing state-of-the-art accelerators. Fig. 16a shows the speedup obtained with different accelerators compared to Intel and Fig. 16b demonstrates the energy reduction. Orianna-OoO achieves 25.6× speedup and 27.5× energy reduction compared to VANILLA-HLS. The reason is that by utilizing factor graph as an abstraction, Orianna-OoO fully exploits the existing sparse structure lies in optimization-based algorithms, avoiding wasting computation on zero elements.

STACK achieves the highest speedup (6.6× compared to Intel) as it stacks three accelerators, each with a tailored architecture designed for individual algorithms. Orianna-OoO achieves comparable performance (only 1% higher latency) compared to STACK but reduces 2.9× energy consumption.

Using one abstraction to accelerate multiple optimization algorithms significantly reduces the hardware resources Orianna consumes. Fig. 16c presents the resource consumption of the three accelerators. Among them, STACK consumes the most resources as it stacks three different accelerators. Although utilizing factor graph as an abstraction, STACK still consumes 3.4× LUT, 3.0× FF, 3.2× BRAM and 2.0× DSP compared to Orianna-OoO. The reason is that with the common abstraction and unified pose representation, Orianna compiler translates different algorithms into the same set of instructions that run on the same accelerator.

7.5 Benefits of Using Factor Graph

One of the major reasons Orianna has better performance and lower energy consumption compared to other robotic accelerators is that we utilize the factor graph abstraction to exploit the existing sparsity and solve the problem incrementally. Thus, a huge sparse matrix decomposition is divided into multiple small yet dense matrix decompositions.

Fig. 17 shows a mobile robot application as an example to demonstrate that Orianna does significantly reduce the size of the matrix operations and Fig. 18 shows that Orianna increases the density of the matrix operations. The original size in localization is 147×90, while the density is only 5.3%.

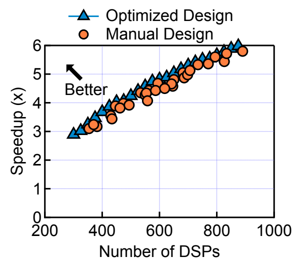


Fig. 19. Speedup comparison between accelerators generated by Orianna and manually designed accelerators over Intel.

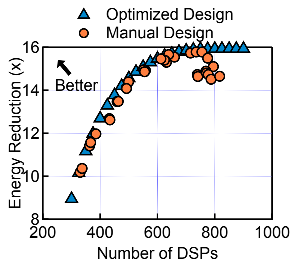


Fig. 20. Energy reduction comparison between accelerators generated by Orianna and manually designed accelerators over Intel.

On average, Orianna reduces the size of matrix operations by 11.1× and improves the density to 58.5%.

The situation is similar in planning and control. When using factor graph as the abstraction, the largest matrix size in planning is only 41 × 12, which is 12.2× smaller compared to VANILLA-HLS, while the density improves by 10.8×. The data is 16.4× and 22.6× for control algorithms.

7.6 Hardware Optimization Analysis

Orianna can generate accelerators under given hardware resource constraints provided by users. With an accurate estimation of the latency and energy consumption of each matrix operation, we formulate the accelerator generation process into a constraint optimization problem. Fig. 19 compares generated accelerators and manually designed accelerators when the number of DSPs is constrained. Orianna shows the best performance under the same constraint.

We further show another example of generating accelerators to minimize energy consumption under specific hardware constraints. Fig. 20 shows a similar trend that Orianna is able to generate accelerators that consume less energy than all other manually designed ones.

8 Related Work

A line of research works has been focusing on accelerating robotic applications. We list a few representative ones in Tbl. 6. There are mainly two directions.

Dedicated robotic accelerators. Prior works on accelerating different robotic algorithms such as localization [3, 5, 15, 17, 21, 30, 34, 35, 56, 57, 66, 69, 70], planning [19, 23, 29, 31, 42–44] and control [20, 32, 68] usually design a dedicated accelerator with emphasis on one or one type of algorithms. Orianna differs from prior works in mainly two aspects. First, Orianna targets to build accelerators for a robotic application that may contain multiple optimization-based algorithms instead of one individual algorithm. Second, Orianna tries to generate an accelerator from the software input of the users instead of manually designing the hardware. Our framework allows users to focus only on high-level options such as the type of sensor inputs and constraints when building up a robotic application.

Table 6. Representative works on robotic accelerators.

Representative works	Application	Algorithm	Generation
RoboX [53]	Drone, Manipulator, Vehicle, Mobile robot	Control	No
Navion [56]	Drone	Localization	No
Hadidi [18]	Drone	Localization	No
Robomorphic [46]	Manipulator	Control	Yes
AutoPilot [24]	Drone	Localization, Planning, Control	Yes
Archytas [36]	Drone, Vehicle	Localization	Yes
Orianna	Drone, Manipulator, Vehicle, Mobile robot	Localization, Planning, Control	Yes

Hardware generation. Hardware accelerator generation is another direction in designing dedicated accelerators [36, 45, 46, 53]. Sharing a similar motivation with us that HLS could not generate high-performance processors in one specific domain, DSAGEN [63] tries to generate spatial architectures where a systolic array accelerator for matrix multiplication is a concrete example. Our work specifically focuses on optimization-based robotic applications, which contain more matrix operations such as rotation matrix transpose and matrix decomposition. We do not generate the detailed architecture for each matrix operation kernel but rather try to find the best combination of them, which allows DSAGEN and our work to be used together. Also, unlike prior works that rely on a data flow graph (DFG) or a macro data flow graph (M-DFG), Orianna uses factor graphs as a common abstraction to generate hardware accelerators. The unique connection between factor graph inference and non-linear optimization solving allows us to translate complicated robotic algorithms into basic matrix operations, thus enabling template-based hardware generation.

9 Conclusion

This paper proposes an accelerator generation framework for robotic applications with multiple optimization-based algorithms. With the proposed unified pose representation, Orianna is able to formulate different robotic algorithms into a factor graph inference and translate them into basic matrix operations. The out-of-order execution of basic instructions on the generated hardware helps Orianna to achieve similar performance compared to state-of-the-art accelerators stacking together, but with much lower hardware resources utilized.

Acknowledgments

The authors would like to thank the National Natural Science Foundation of China under Grant U21B2031 for their support.

References

[1] Daniel Albiero, Angel Pontin Garcia, Claudio Kiyoshi Umezu, and Rodrigo Leme de Paulo. Swarm robots in mechanized agricultural operations: A review about challenges for research. *Computers and Electronics in Agriculture*, 193:106608, 2022.

[2] Kostas Alexis, Christos Papachristos, George Nikolakopoulos, and Anthony Tzes. Model predictive quadrotor indoor position control. In *2011 19th Mediterranean Conference on Control & Automation (MED)*, pages 1247–1252. IEEE, 2011.

[3] Bahar Asgari, Ramyad Hadidi, Nima Shoghi Ghaleshahi, and Hyesoon Kim. Pisces: power-aware implementation of slam by customizing efficient sparse algebra. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[4] José Luis Blanco-Claraco. A tutorial on $\mathbf{SE}(3)$ transformation parameterizations and on-manifold optimization. *arXiv preprint arXiv:2103.15980*, 2021.

[5] Konstantinos Boikos and Christos-Savvas Bouganis. Semi-dense slam on an fpga soc. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[6] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization.* Cambridge university press, 2004.

[7] Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis.* Cengage learning, 2015.

[8] Peter Corke. *Robotics and control: fundamental algorithms in MATLAB®*, volume 141. Springer Nature, 2021.

[9] Frank Dellaert. Factor graphs: Exploiting structure in robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 4:141–166, 2021.

[10] Frank Dellaert and Michael Kaess. Square root sam: Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, 25(12):1181–1203, 2006.

[11] Frank Dellaert, Michael Kaess, et al. Factor graphs for robot perception. *Foundations and Trends® in Robotics*, 6(1-2):1–139, 2017.

[12] Ethan Eade. Lie groups for 2d and 3d transformations. *URL http://ethaneade. com/lie. pdf, revised Dec*, 117:118, 2013.

[13] Farbod Fahimi. *Autonomous robots.* Springer, 2009.

[14] Joel N Franklin. *Matrix theory.* Courier Corporation, 2012.

[15] Yiming Gan, Yu Bo, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang, and Yuhao Zhu. Eudoxus: Characterizing and accelerating localization in autonomous machines industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 827–840. IEEE, 2021.

[16] Xiang Gao and Tao Zhang. *Introduction to visual SLAM: from theory to practice.* Springer Nature, 2021.

[17] Quentin Gautier, Alric Althoff, and Ryan Kastner. Fpga architectures for real-time dense slam. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160, pages 83–90. IEEE, 2019.

[18] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. Quantifying the design-space tradeoffs in autonomous drones. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 661–673, 2021.

[19] Yuhui Hao, Yiming Gan, Bo Yu, Qiang Liu, Shao-Shan Liu, and Yuhao Zhu. Blitzcrank: Factor graph accelerator for motion planning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[20] Yuhui Hao, Bo Yu, Qiang Liu, and Shao-Shan Liu. Fglqr: Factor graph accelerator of lqr control for autonomous machines. *arXiv preprint arXiv:2308.02768*, 2023.

[21] Yuhui Hao, Bo Yu, Qiang Liu, Shaoshan Liu, and Yuhao Zhu. Factor graph accelerator for lidar-inertial odometry. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–7, 2022.

[22] Philipp Junietz, Farid Bonakdar, Björn Klamann, and Hermann Winner. Criticality metric for the safety validation of automated driving using model predictive trajectory optimization. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 60–65. IEEE, 2018.

[23] Atsutake Kosuge and Takashi Oshima. A 1200× 1200 8-edges/vertex fpga-based motion-planning accelerator for dual-arm-robot manipulation systems. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2. IEEE, 2020.

[24] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 300–317. IEEE, 2022.

[25] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[26] F Künhe, J Gomes, and W Fetter. Mobile robot trajectory tracking using model predictive control. In *II IEEE latin-american robotics symposium*, volume 51, 2005.

[27] Georgia Tech Borg Lab. Gtsam. https://github.com/borglab/gtsam. Accessed: 2023-07-1.

[28] Forrest Laine and Claire Tomlin. Efficient computation of feedback control for equality-constrained lqr. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6748–6754. IEEE, 2019.

[29] Ruige Li, Xiangcai Huang, Sijia Tian, Rong Hu, Dingxin He, and Qiang Gu. Fpga-based design and implementation of real-time robot motion planning. In *2019 9th International Conference on Information Science and Technology (ICIST)*, pages 216–221. IEEE, 2019.

[30] Ziyun Li, Yu Chen, Luyao Gong, Lu Liu, Dennis Sylvester, David Blaauw, and Hun-Seok Kim. An 879gops 243mw 80fps vga fully visual cnn-slam processor for wide-range autonomous exploration. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 134–136. IEEE, 2019.

[31] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[32] I-Ting Lin, Zih-Sing Fu, Wen-Ching Chen, Liang-Yi Lin, Nian-Shyang Chang, Chun-Pin Lin, Chi-Shi Chen, and Chia-Hsiang Yang. 2.5 a 28nm 142mw motion-control soc for autonomous mobile robots. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 1–3. IEEE, 2023.

[33] Qiang Liu, Yuhui Hao, Weizhuang Liu, Bo Yu, Yiming Gan, Jie Tang, Shao-Shan Liu, and Yuhao Zhu. An energy efficient and runtime reconfigurable accelerator for robotic localization. *IEEE Transactions on Computers*, 2022.

[34] Qiang Liu, Shuzhen Qin, Bo Yu, Jie Tang, and Shaoshan Liu. π-ba: Bundle adjustment hardware accelerator based on distribution of 3d-point observations. *IEEE Transactions on Computers*, 69(7):1083–1095, 2020.

[35] Runze Liu, Jianlei Yang, Yiran Chen, and Weisheng Zhao. eslam: An energy-efficient accelerator for real-time orb-slam on fpga platform. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[36] Weizhuang Liu, Bo Yu, Yiming Gan, Qiang Liu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 479–493, 2021.

[37] Shaoshuai Luo, Xiaohui Li, and Zhenping Sun. An optimization-based motion planning method for autonomous driving vehicle. In *2020 3rd International Conference on Unmanned Systems (ICUS)*, pages 739–744. IEEE, 2020.

[38] Sean Mason, Nicholas Rotella, Stefan Schaal, and Ludovic Righetti. Balancing and walking using full dynamics lqr control with contact constraints. In *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pages 63–68. IEEE, 2016.

[39] Mustafa Mukadam, Jing Dong, Frank Dellaert, and Byron Boots. Simultaneous trajectory estimation and planning via probabilistic inference. In *Robotics: Science and systems*, 2017.

[40] Mustafa Mukadam, Jing Dong, Xinyan Yan, Frank Dellaert, and Byron Boots. Continuous-time gaussian process motion planning via probabilistic inference. *The International Journal of Robotics Research*, 37(11):1319–1340, 2018.

[41] Richard M Murray, Zexiang Li, and S Shankar Sastry. *A mathematical introduction to robotic manipulation*. CRC press, 2017.

[42] Sean Murray, Will Floyd-Jones, George Konidaris, and Daniel J Sorin. A programmable architecture for robot motion planning acceleration. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160, pages 185–188. IEEE, 2019.

[43] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J Sorin, and George Dimitri Konidaris. Robot motion planning on a chip. In *Robotics: Science and Systems*, volume 6, 2016.

[44] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[45] Sabrina M Neuman, Radhika Ghosal, Thomas Bourgeat, Brian Plancher, and Vijay Janapa Reddi. Roboshape: Using topology patterns to scalably and flexibly deploy accelerators across robots. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[46] Sabrina M Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: a design methodology for domain-specific accelerators parameterized by robot morphology. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 674–686, 2021.

[47] Nvidia. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2023-12-1.

[48] Nvidia. cuSOLVER. https://docs.nvidia.com/cuda/cusolver/index.html. Accessed: 2023-12-1.

[49] Nvidia. TX1 datasheet. http://images.nvidia.com/content/tegra/embedded-systems/pdf/JTX1-Module-Product-sheet.pdf. Accessed: 2023-07-1.

[50] Fauzi Othman, MA Bahrin, N Azli, et al. Industry 4.0: A review on industrial automation and robotic. *J Teknol*, 78(6-13):137–143, 2016.

[51] Kanubhai K Patel. Design of efficient intelligent autonomous surface cleaner. In *2022 IEEE World Conference on Applied Intelligence and Computing (AIC)*, pages 103–108. IEEE, 2022.

[52] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018.

[53] Jacob Sacks, Divya Mahajan, Richard C Lawson, Behnam Khaleghi, and Hadi Esmaeilzadeh. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 479–490. IEEE, 2018.

[54] Tixiao Shan, Brendan Englot, Drew Meyers, Wei Wang, Carlo Ratti, and Daniela Rus. Lio-sam: Tightly-coupled lidar inertial odometry via smoothing and mapping. In *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 5135–5142. IEEE, 2020.

[55] Joan Sola, Jeremie Deray, and Dinesh Atchuthan. A micro lie theory for state estimation in robotics. *arXiv preprint arXiv:1812.01537*, 2018.

[56] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones. *IEEE Journal of Solid-State Circuits*, 54(4):1106–1119, 2019.

[57] Rongdi Sun, Peilin Liu, Jianwei Xue, Shiyu Yang, Jiuchao Qian, and Rendong Ying. Bax: A bundle adjustment accelerator with decoupled access/execute architecture for visual odometry. *IEEE Access*, 8:75530–75542, 2020.

[58] Duy-Nguyen Ta, Marin Kobilarov, and Frank Dellaert. A factor graph approach to estimation and model predictive control on unmanned aerial vehicles. In *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 181–188. IEEE, 2014.

[59] Mahdi Tavakoli, Jay Carriere, and Ali Torabi. Robotics, smart wearable technologies, and autonomous intelligent systems for healthcare during the covid-19 pandemic: An analysis of the state of the art and future vision. *Advanced Intelligent Systems*, 2(7):2000071, 2020.

[60] Zishen Wan, Ashwin Lele, Bo Yu, Shaoshan Liu, Yu Wang, Vijay Janapa Reddi, Cong Hao, and Arijit Raychowdhury. Robotic computing on fpgas: Current progress, research challenges, and opportunities. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 291–295. IEEE, 2022.

[61] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. A survey of fpga-based robotic computing. *IEEE Circuits and Systems Magazine*, 21(2):48–74, 2021.

[62] Zishen Wan, Yuyang Zhang, Arijit Raychowdhury, Bo Yu, Yanjun Zhang, and Shaoshan Liu. An energy-efficient quad-camera visual system for autonomous machines on fpga platform. In *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 1–4. IEEE, 2021.

[63] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[64] Xilinx. Xilinx zynq-7000 soc zc706 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html. Accessed: 2023-07-1.

[65] Shuo Yang, Gerry Chen, Yetong Zhang, Howie Choset, and Frank Dellaert. Equality constrained linear optimal control with factor graphs. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9717–9723. IEEE, 2021.

[66] Jae-Sung Yoon, Jeong-Hyun Kim, Hyo-Eun Kim, Won-Young Lee, Seok-Hoon Kim, Kyusik Chung, Jun-Seok Park, and Lee-Sup Kim. A graphics and vision unified processor with 0.89 μw/fps pose estimation engine for augmented reality. In *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 336–337. IEEE, 2010.

[67] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and systems*, pages 1–9. Berkeley, CA, 2014.

[68] Pei Zhang, Aaron Mills, Joseph Zambreno, and Phillip H Jones. The design and integration of a software configurable and parallelized coprocessor architecture for lqr control. *Journal of Parallel and Distributed Computing*, 106:121–131, 2017.

[69] Zhe Zhang, Shaoshan Liu, Grace Tsai, Hongbing Hu, Chen-Chi Chu, and Feng Zheng. Pirvs: An advanced visual-inertial slam system with flexible sensor fusion and hardware co-design. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3826–3832. IEEE, 2018.

[70] Zhengdong Zhang, Amr AbdulZahir Suleiman, Luca Carlone, Vivienne Sze, and Sertac Karaman. Visual-inertial odometry on chip: An algorithm-and-hardware co-design approach. In *Proceedings of Robotics Science and Systems (RSS)*, 2017.

[71] Chengmin Zhou, Bingding Huang, and Pasi Fränti. A review of motion planning algorithms for intelligent robots. *Journal of Intelligent Manufacturing*, 33(2):387–424, 2022.