An Energy Efficient and Runtime Reconfigurable Accelerator for Robotic Localization

Qiang Liu, *Member, IEEE*, Yuhui Hao, Weizhuang Liu, Bo Yu, *Senior Member, IEEE*, Yiming Gan, Jie Tang, *Senior Member, IEEE*, Shao-Shan Liu, *Senior Member, IEEE*, and Yuhao Zhu, *Member, IEEE*

Abstract—Accurate and efficient localization of robots under limited on-board resources has fueled specialized localization accelerators. Despite many recent efforts, accelerating robotic localization is still fundamentally challenging. To tackle the challenges, the paper proposes a configurable hardware architecture and a design space optimization method to automatically generate an optimal accelerator design under the design constraints. Data locality, sparsity, and fixed-point arithmetic optimization techniques that are specific to the localization algorithm are exploited to customize the accelerator. In addition, a low-cost runtime configuration mechanism is proposed to enable the accelerator to continuously optimize itself at runtime according to the operating environment to save power while sustaining performance and accuracy. The evaluation on FPGA demonstrates that the proposed accelerator achieves orders of magnitude performance improvement and/or energy savings compared to the software implementation on Intel and Arm CPUs; and substantially outperforms existing FPGA accelerators in terms of performance and energy.

1 Introduction

Autonomous machines such as drones, logistic robots, and self-driving cars are becoming an integral part of our everyday life [1]. An autonomous machine localizes itself by calculating its position in an environment. Since the environment in many scenarios is unknown, the agent usually simultaneously constructs a map of the environment while localizing itself, giving rise to the notion of Simultaneous Localization and Mapping (SLAM) [2].

Principled mathematical solutions for SLAM include filtering-based [3] and non-linear optimization-based [2], [4], where the latter recently shows higher robustness but with intensive computation. Given the limited computing capability, memory space and power supply on mobile devices, existing SLAM systems run anywhere between 2 FPS and 20 FPS, far from the real-time process requirement [5], and contribute to 45% of the system power consumption [6].

While there exists a great amount of recent efforts focus on accelerating various robotics tasks [7], [8], [9], [10], [11] on portable devices, accelerating localization has three challenges. First, to arrive at a practical design, designers often must explore a large design space considering performance, resource and power constraints and the localization accuracy. Second, the environment, in which an autonomous machine operates, is constantly changing, and thus the load of work at runtime is dynamically changing. A static accelerator, as in virtually all prior work, accommodates the worst case, necessarily leading to wasteful computation and energy. Third, principled localization software typically employs numerical optimization algorithms based on double precision arithmetic operations to find stable and precision solutions. Implementing a large amount of double precision arithmetic operations by hardware is prohibitively expensive in terms of hardware resource and power consumption.

In this work, we present an energy-efficient and runtime-reconfigurable FPGA accelerator for robotic localization. By exploiting SLAM-specific data locality, sparsity and dependency, we design efficient computing and memory architecture with data reuse, resource sharing, and operation pipelining. Specifically, we identify a set of reconfigurable hardware blocks whose designs dictate the trade-off among resource, latency and power consumption. Given the design specifications (e.g., resource and latency constraints), the design space exploration is formulated as a constrained optimization problem and the optimized configurations of the reconfigurable blocks are generated by solving the problem. To further reduce the hardware cost, we use fixed-point arithmetic and reduced bitwidth without sacrificing localization accuracy.

Moreover, we leverage the observation that the load of work at runtime varies with the environment. The accelerator can be reconfigured at runtime according to the workloads to further save power consumption. Based on the proposed configurable hardware blocks, our previously developed synthesis framework [12] can automatically

- *Qiang Liu, Yuhui Hao, and Weizhuang Liu are with the Tianjin Key Laboratory of Imaging and Sensing Microelectronic Technology, School of Microelectronics, Tianjin University, Tianjin 300072, China. E-mail: {qiangliu, yuhuihao, liuwz}@tju.edu.cn.*
- *Bo Yu and Shao-Shan Liu are with the BeyonCa, Houston, TX 77002 USA. E-mail: {bo.yu, shaoshan.liu}@beyonca.com.*
- *Yiming Gan and Yuhao Zhu are with the University of Rochester, Rochester, NY 14642 USA. E-mail: ygan10@ur.rochester.edu, yzhu@rochester.edu.*
- *Jie Tang is with the South China University of Technology, Guangzhou 510641, China. E-mail: cstangjie@scut.edu.cn.*

Manuscript received 22 February 2022; revised 6 December 2022; accepted 16 December 2022. Date of publication 20 December 2022; date of current version 9 June 2023.
This work was supported in part by the National Natural Science Foundation of China under Grants 61974102 and U21B2031.
(Corresponding authors: Bo Yu, Jie Tang and Yuhao Zhu.)
Recommended for acceptance by R. Wang.
Digital Object Identifier no. 10.1109/TC.2022.3230899

generate localization accelerators from high-level algorithm descriptions, using the block design container-like method. This paper builds on the basic hardware design in [12] and carries out more optimization on microarchitecture design.

In summary, this work makes the following contributions:

- We propose an energy efficient SLAM backend accelerator, in which the key blocks are designed with configurability and enable trade-off among resource, latency and power consumption. The design space exploration is formulated as a convex optimization problem, solving which gives the optimized accelerator configuration. We demonstrate that the accelerators can flexibly trade performance for energy and out-perform existing localization accelerators.
- We perform multiple algorithmic, numeric precision and microarchitectural optimization by exploiting the data sparsity, locality, distribution and parallelism inherent in SLAM, leading to low computation complexity, high pipeline throughput and reduced memory space.
- We propose a surrounding-ware configuration mechanism that combined with low-power technology such as clock-gating adjusts hardware at runtime to save power according to the operating environments without degrading performance.

The rest of this paper is organized as follows. Section 2 gives an overview of the SLAM problem and the algorithm. Section 3 presents the accelerator architecture in details. Section 4 describes the numeric precision optimization method for the arithmetical operations of the accelerator. Section 5 formulates the architectural design space exploration of the accelerator into a constrained optimization problem, based on which a runtime configuration mechanism is proposed. Section 6 evaluates the proposed accelerator in terms of speed, energy, resource usage and runtime efficiency. Section 7 introduces some existing localization accelerators related to this work and is followed by the conclusion in Section 8.

2 Background

We first introduce the localization problem (Section 2.1), and then describe the most widely used solving algorithm (Section 2.2), which we target in this paper. The descriptions show the computation complexity and features of SLAM.

2.1 SLAM Problem Formulation

While other localization settings are possible, this paper focuses on SLAM since it is the least restrictive and is widely used in autonomous machines, such as self-driving cars [11] and drones [7], [10].

Formally, a 3 dimensional (3D) map is a set of points in the world coordinate system, where each point is represented by the $< x, y, z >$ coordinates. Localization generates the pose of an agent in the world coordinate system. The pose is represented by the six degrees of freedom (DoF), which includes the three *translational* DoF, i.e., the $p =< x, y, z >$ coordinates, and the three *rotational* DoF, i.e., the orientation q about the three orthogonal axes (a.k.a., yaw, roll, and pitch).

Maximum a posteriori (MAP) estimation is an efficient approach to estimate the pose [2]. Comparing to the popular class of SLAM algorithm based on non-linear filtering [3], MAP is more robust in long-term localization [4].

MAP for robotics has two key features [2]. First, MAP in robotics often fuses measurements from multiple sensors such as camera and Inertial Measurement Unit (IMU) to improve the estimation accuracy. Our work targets a common camera+IMU setup. Second, only a fixed number of keyframes of measurements, maintained through a sliding window, are used for MAP estimation to ensure real-time performance [13]. As the window slides, the oldest measurements are moved out of the window. These measurements, then become the *prior* in the next MAP estimate, through a process known as marginalization [14], to constrain and guide the state estimation for the next window.

More formally, given the sensor measurements in a window, the goal of MAP is to estimate a state vector $\mathbf{p}$, which contains the sequence of the machine's poses s (i.e., localization) and the 3D coordinates λ of the points in space captured by the camera (i.e., mapping) in the current window

$$\mathbf{p} = [s_1, \ldots, s_i, \ldots, s_n, \lambda_1, \ldots, \lambda_j, \ldots \lambda_m], \tag{1}$$

where s_i represents the pose of the robot at the i-th measurement, and λ_j represents the 3D coordinates of the j-th feature (landmark) point. In addition to the 6 DoF pose p and q, with IMU, the pose is also represented by the line velocity v, the accelerometer bias b^a and the gyroscope bias b^g in the IMU three orthogonal axes. Note that p, q, v, b^a and b^g are 3D vectors $\mathcal{R}^3$.

To estimate $\mathbf{p}$, MAP solves a nonlinear least squares (NLS) optimization problem as below [15]

$$\min_{\mathbf{p}} \left\{ \sum_{i=1}^{N} |\mathbf{o}_i - \mathcal{P}_i(\mathbf{p})|^2_{\mathbf{C}_i} + |\mathbf{r}_p - \mathbf{H}_p \mathbf{p}|^2 \right\}, \tag{2}$$

where N is the number of sensors; $\mathbf{o}_i$ is the actual measurement of the i-th sensor ("ground truth"), $\mathcal{P}_i$ is a function that maps the (to-be-estimated) state vector $\mathbf{p}$ to the i-th sensor's measurement space[1], $\mathbf{C}_i$ is the covariance matrix of the i-th sensor, and $|\cdot|^2_{\mathbf{C}}$ is the Mahalanobis norm that quantifies the error in the current window; $\mathbf{r}_p$ and $\mathbf{H}_p$ are the prior and roughly correspond to the marginalized measurement and covariance matrix [14], respectively, and $|\cdot|^2$ (L2 norm) quantifies the errors imposed by the priors.

2.2 Problem Solving Algorithm

Fig. 1 shows the overall structure of the MAP estimation algorithm, which mainly consists of two phases: 1) a NLS solver that solves (2) for the current sliding window, and 2) marginalization, which generates the prior to form the NLS objective function of the next window. These two phases are executed in sequence.

NLS Solver. The Levenberg-Marquardt (LM) algorithm [16] is a widely-used NLS solver. It belongs to the class of gradient

1. For instance, for camera measurements, $\mathcal{P}$ is a 3D to 2D camera projection.

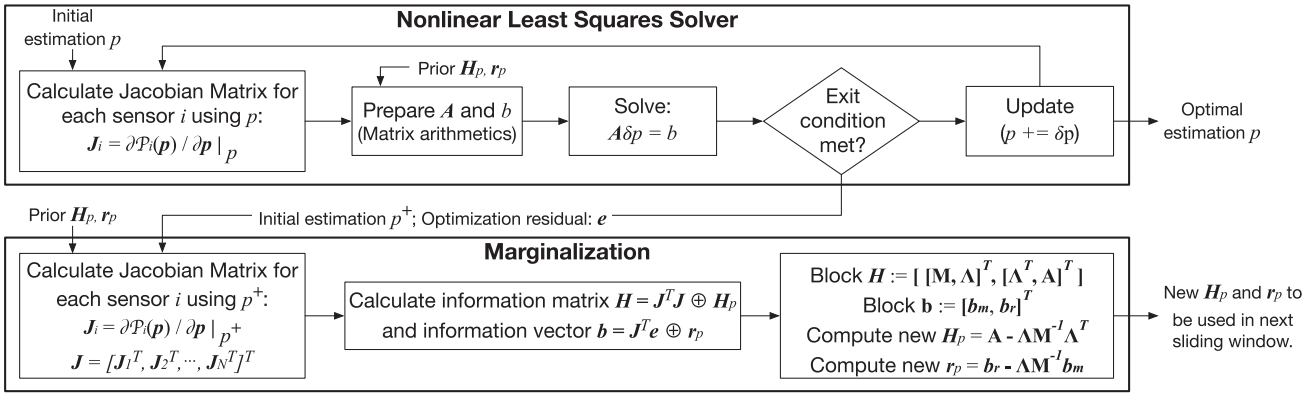


Fig. 1. Algorithm overview of MAP estimation.

descent-based algorithms that iteratively update the result $\mathbf{p}$ with a new estimate $\mathbf{p} + \delta\mathbf{p}$ until the final result $\mathbf{p}^+$ is calculated. Each iteration calculates $\delta\mathbf{p}$ in three steps:

1) Calculate the Jacobian matrix (the matrix of all the partial derivatives of the objective function) using $\mathbf{p}$;
2) Given the Jacobian matrix and the priors $\mathbf{H}_p$ and $\mathbf{r}_p$, form a system of linear equations $\mathbf{A}\delta p = \mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are calculated through a sequence of matrix operations.
3) Solve the linear system to obtain $\delta\mathbf{p}$.

Marginalization. Marginalization uses the output of NLS solver $\mathbf{p}^+$, i.e., the estimated state of the current window, to generate the *priors*, $\mathbf{H}_p$ and $\mathbf{r}_p$, which will participate in the objective function of the next window. The priors are generated in three steps [14]:

1) Calculate the Jacobian matrix $\mathbf{J}$ and the residual ϵ (error of the loss function) using $\mathbf{p}^+$;
2) Calculate the *information matrix* $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ and the *information vector* $\mathbf{b} = \mathbf{J}^T\epsilon$.
3) Block $\mathbf{H}$ as $\begin{bmatrix} \mathbf{M} & \Lambda^T \\ \Lambda & \mathbf{A} \end{bmatrix}$ and block $\mathbf{b}$ as $\begin{bmatrix} \mathbf{b}_m \\ \mathbf{b}_r \end{bmatrix}$. Calculate the new priors using Schur complement [14]: $\mathbf{H}_p = \mathbf{A} - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{\Lambda}^T$, and $\mathbf{r}_p = \mathbf{b}_r - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{b}_m$.

3 Customizable Hardware

This section first gives the overview of the overall hardware architecture (Section 3.1). Then, customization of the key blocks of the architecture by leveraging the data and computation patterns inherent to SLAM is presented (Section 3.2 to Section 3.7). Multiple algorithmic and microarchitectural optimizations are carried out in different blocks.

3.1 Hardware Design Overview

Fig. 2 shows the overall hardware architecture, which accelerates both NLS solver and marginalization phases. The NLS solver circuit first calculates Jacobians of sensor measurements. Here, it shows visual and IMU measurements and more other sensor measurements could be added. Schur elimination is performed on the visual Jacobain to simplify the subsequent computation and storage. Then, the matrix and vector of the system of linear equations are formed with on-chip storage optimization. Finally, Cholesky decomposition is performed to solve the linear system.

Marginalization is performed after NLS solver due to data dependency. Therefore, both phases share certain hardware blocks. In Fig. 2, solid lines denote the NLS solver data flow, and dash lines denote the marginalization data flow.

We identify a set of customizable hardware blocks, indicated by gears in Fig. 2. Specifically, the customizable blocks are the Schur elimination block, the Cholesky decomposition block, and the Marginalization block. These blocks present a large resource-vs-performance trade-off space. Preliminary result in [12] shows that varying the design parameters of the blocks could affect the end-to-end latency by over 20× and the overall resource consumption by about 3×.

While many blocks in the hardware architecture have well-established implementations (e.g., dense matrix multiplication), we now describe the designs of the blocks which are optimized by harnessing the data and computation features inherent in SLAM.

3.2 Visual Jacobian Block

The visual Jacobian block computes visual residual $\epsilon_c = \mathbf{o}_c - \mathcal{P}_c(\mathbf{p})$ and Jacobian $\mathbf{J}_c$. Different from [9], the visual Jacobian block uses analytic differentiation to compute the partial derivatives for fast computation. The detailed architecture design of the visual Jacoban block is shown in Fig. 3, based on a set of configurable basic computing units in Fig. 4. The MAC unit can perform matrix-matrix multiplication and matrix-vector multiplication. The QM unit can perform multiplication of two Quaternions and multiplication of a Quaternion and a Quaternion' conjugate. The CTU unit performs coordinate system conversion.

Let us explain the hardware design using a simple example in Fig. 5. In the example a sliding window consisting of 2 keyframes, 3 feature points ($P_1 \sim P_3$), and 4 observations ($O_1 \sim O_4$) is shown. Recall that the observations are generated from feature points through the projection function ($\mathcal{P}$ in (2)). Because the Jacobian matrix is formed by the partial derivatives of $\mathcal{P}$ with respect to the valid $<$feature point, observation$>$ pairs, the 3×4 matrix $\mathbf{J}_c$ with sparsity (shaded blocks indicate non-zero values) is shown in Fig. 5.

We divide the computation into three levels: keyframe, feature and observation, as shown in Fig. 3. The Observation block is responsible for calculating the matrix elements (partial derivatives), which requires two pieces of data: 1)

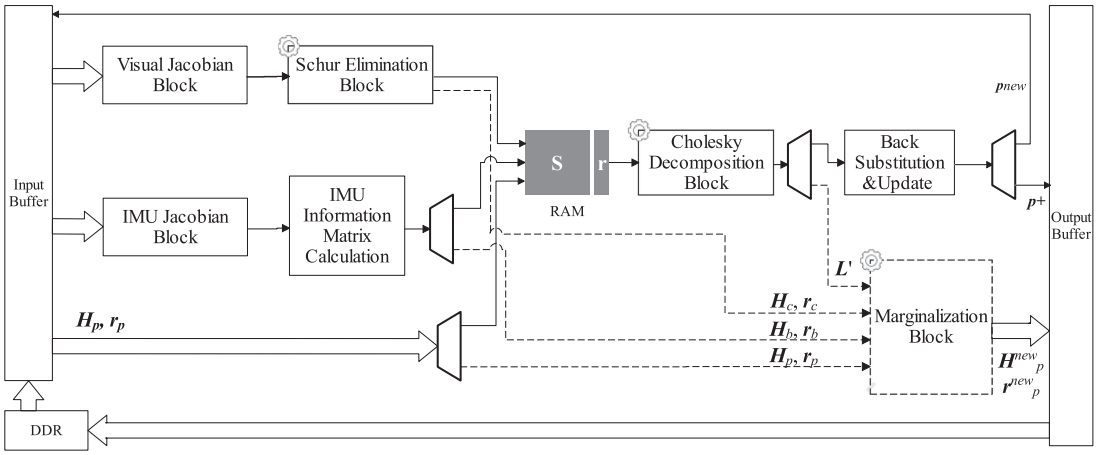


Fig. 2. Overall hardware architecture. Three blocks, Schur elimination, Cholesky Decomposition, and Marginalization, are parameterized. Solid lines denote the NLS solver data flow, and dash lines denote the marginalization data flow.

the coordinates of the feature point, e.g., P_1, in the world coordinate system, the IMU coordinate system and the camera coordinate system; this is calculated by the Feature block; and 2) the rotation matrices of keyframes, which contain the observations of the feature point (e.g., keyframes 1 and 2 which containing O_1 and O_3 of P_1); this is calculated by the Keyframe block. The three level division allows us to exploit characteristics unique to SLAM data to effectively capture data reuse and operation pipeline.

Data Reuse. Two forms of data-reuse exist. First, each feature point is reused across its associated observations (e.g., P_1 is reused over O_1 and O_3). Second, each keyframe's rotation matrix is reused over all the observations within the keyframe (e.g., keyframe 1's rotation matrix is reused over O_1 and O_2). We decide to exploit feature point reuse, due to the fact that the number of feature points is far more than the number of keyframes. In our profiling, a typical sliding window on average would have $10\times$ more feature points than keyframes. This decision leads to design a "feature-stationary" architecture where each feature point stays in the Observation block until an entire matrix row is calculated.

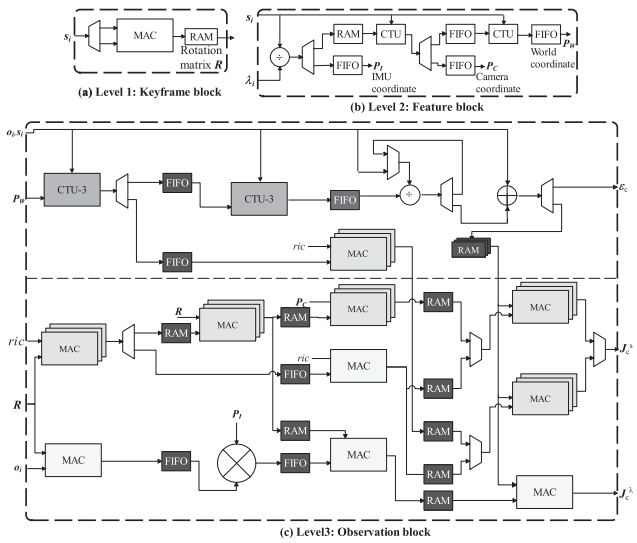


Fig. 3. Microarchitecture of the three blocks in the visual Jacobian hardware. CTU-3: 3-stage pipelined CTU.

Given the "feature-stationary" design, the Feature block and the Observation block form a producer-consumer pair. As a result, their communication (coordinates P_I, P_C and P_W) is through FIFOs as shown in Fig. 3. In contrast, accesses to keyframe rotation matrices are arbitrary, since different observations of a feature point could be captured in different keyframes. Thus, the rotation matrices R are stored in a RAM. Note that a RAM is much more power-hungry to access than a FIFO.

Operation Pipeline. With the Feature block and the Observation block communicating through a FIFO, it is critical to ensure that speeds of the two blocks roughly match so as to avoid large FIFOs or constantly stalling the pipeline. We observe that the number of observations is typically $10\times$ more than that of feature points. Thus, the Observation block does more computation, and needs to be more aggressively pipelined.

Because the amount of work the Observation block processes for each feature point varies, we statically pipeline the blocks based on the *average* statistics profiled offline, to simplify the hardware design. In particular, we first pipeline the Observation block as deep as possible to maximize the throughput, and denote the per-stage latency C_o. Fig. 3c shows the pipelined architecture of the Observation block with 8 pipeline stages. We then estimate the average number of observations associated with a feature point, and denote it N_o. Given N_o and C_o, the Feature block is then pipelined into $L_f/(N_oC_o)$ stages, where L_f denotes the latency of the Feature block and is fixed since the amount of work the Feature block does for each feature point is constant. To make the pipeline of the Feature block is adjustable, a configurable CTU unit is designed in Fig. 4c and its pipeline stage can be set to 1, 2 with FIFO_1 or 3 with FIFO_2 and QM_2.

3.3 IMU Jacobian Block

The IMU Jacobian block computes the IMU residual ϵ_b and Jacobian matrix J_b. J_b is the matrix of partial derivatives of ϵ_b with respective to pose s_i. According to the meaning of ϵ_b and s_i, J_b is expressed in a blocked matrix to reveal the sparsity, as shown in Fig. 6. Each block indicates the partial derivative of ϵ_b with respective to each element of s_i, e.g., B_1 is the partial derivative of ϵ_b over q_i.

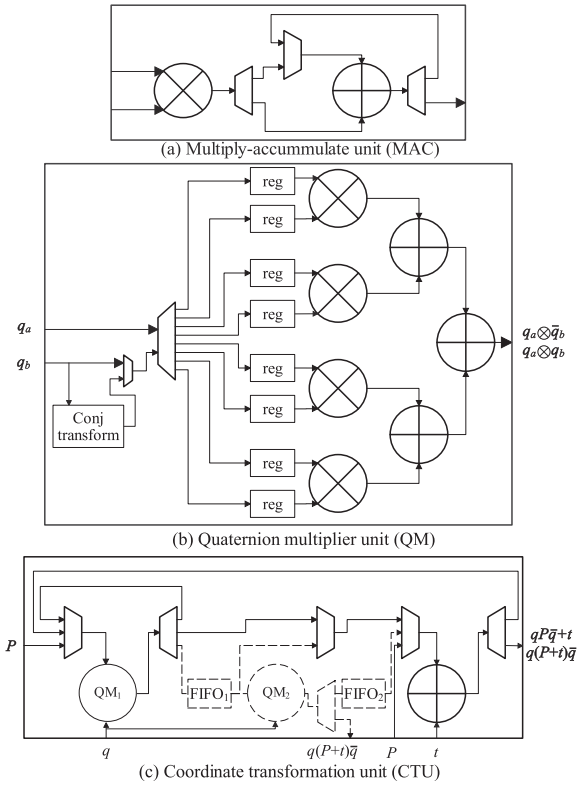


Fig. 4. Configurable basic computing units.

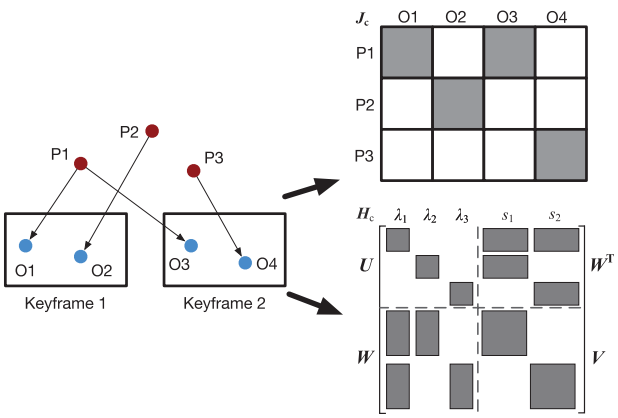


Fig. 5. Illustration of a visual Jacobian matrix (3×4) and a information matrix (33×33) for a simple sliding window with 2 keyframes, 3 feature points ($P_1 \sim P_3$), and 4 observations ($O_1 \sim O_4$).

Fig. 7 shows the detailed structure of the IMU Jacobian block. We reorganize the computations for two purposes: 1) extract common computations to avoid repeated calculation of intermediate values, such as p_{mid}, q_{mid} and v_{mid}, and 2) map the computations onto the configurable basic computing units. J_b and ϵ_b are computed sequentially in two pipeline stages in Fig. 7a. Stage 1 computes subblocks of J_b in parallel as shown in Fig. 7b, and stage 2 computes ϵ_b and forms J_b. The parallel structure of stage 1 ensures the latency matching of two stages.

3.4 Linear Equation Simplification Using Schur Elimination

The main complexity in the NLS solver is to solve the linear system $\mathbf{A}\delta\mathbf{p} = \mathbf{b}$, where $\mathbf{A}$ is a $(p+q) \times (p+q)$ matrix, and

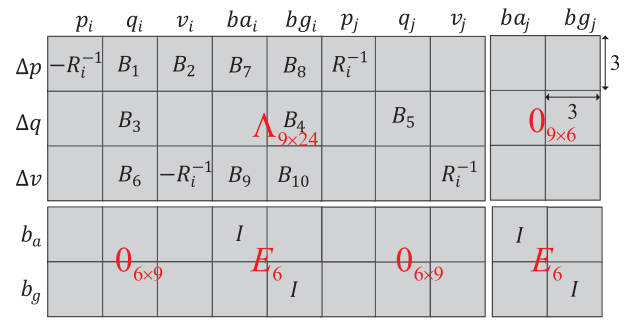


Fig. 6. Blocked format of the IMU Jacobian.

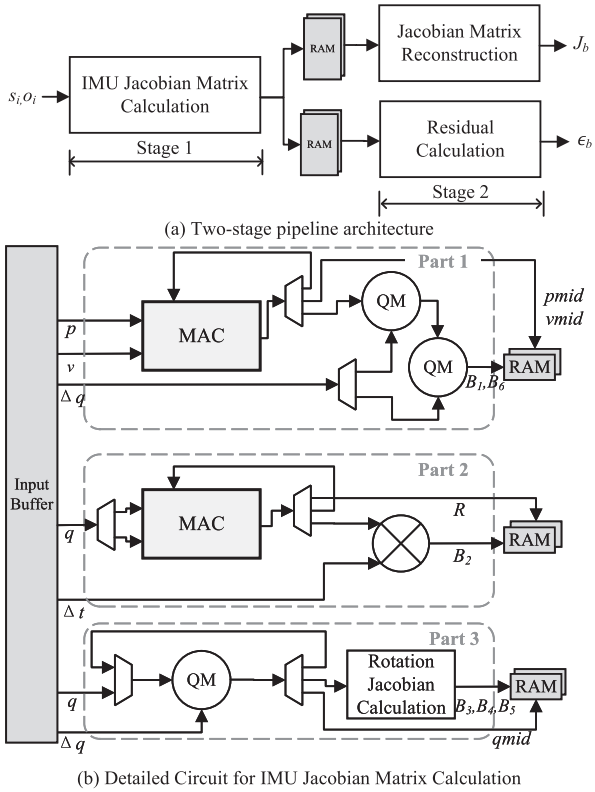


Fig. 7. Microarchitecture of the IMU Jacobian block.

both $\delta\mathbf{p}$ and $\mathbf{b}$ are $(p+q)$-dimensional column vectors. The linear system can be transformed to two simpler systems of linear equations, each with a size of $p \times p$ and $q \times q$, respectively, that are cheaper to solve, using Schur elimination [17].

For the SLAM, $\mathbf{A} = \mathbf{H} + \mu\mathbf{I}$ is a $(15n+m) \times (15n+m)$ matrix. Specifically, the information matrix $\mathbf{H}$ is composed of three parts: visual information matrix $\mathbf{H}_{c_{(15n+m)\times(15n+m)}}$, IMU information matrix $\mathbf{H}_{b_{15n\times15n}}$ and the prior $\mathbf{H}_{p_{15n\times15n}}$. Note that the IMU information matrix and the prior matrix only contain the pose information. Therefore, Schur elimination is first applied to the visual part to simplify the linear system by separating pose s and feature point coordinate λ.

Fig. 5 shows the blocked form of $\mathbf{H}_c = \begin{bmatrix} \mathbf{U} & \mathbf{W}^T \\ \mathbf{W} & \mathbf{V} \end{bmatrix}$ of the example. Generally, $\mathbf{U}$, $\mathbf{W}$ and $\mathbf{V}$ are $m \times m$, $15n \times m$ and $15n \times 15n$ matrices. The original linear system can then be expressed in the following blocked-matrix form

$$\begin{cases} \mathbf{U}\delta\mathbf{p}_x + \mathbf{W}^T\delta\mathbf{p}_y & = \mathbf{b}_x \\ (\mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{W}^{\mathbf{T}})\delta\mathbf{p}_y & = \mathbf{b}_y - \mathbf{W}\mathbf{U}^{-1}\mathbf{b}_x \end{cases} \tag{3}$$

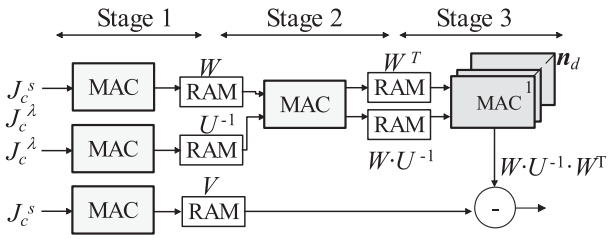


Fig. 8. Microarchitecture of the Schur elimination block.

Critically, the second half of (3) is a $15n \times 15n$ system and requires solving only $\delta \mathbf{p}_y$, solving which would allow us to solve the first half of (3), a $m \times m$ system that involves only $\delta \mathbf{p}_x$. We define $\mathbf{S} = \mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{W}^{\mathbf{T}}$ and $\mathbf{r} = \mathbf{b}_y - \mathbf{W}\mathbf{U}^{-1}\mathbf{b}_x$. Then the linear system is reconstructed as $\mathbf{S}\delta\mathbf{p}_y = \mathbf{r}$.

The Schur elimination (SE) block and the IMU information matrix computation (IMC) block in Fig. 2 generate the visual part and the IMU part of $\mathbf{S}$ and $\mathbf{r}$, respectively.

Fig. 8 shows the detailed architecture of the SE block. The Schur elimination of different feature points are independent. Therefore, different from [9], the SE block is pipelined across the feature points in three stages for simplified hardware design. The main computations are matrix multiplications, which are mapped on the configurable MAC units. The latency of each pipeline stage depends on the number of observations N_0. The latency of three stages is $12N_0$, $6N_0$ and $(6N_0)^2$, respectively. Stage 3 has the longest latency. To improve the pipeline throughput, we increase the number of MAC units $\mathbf{n}_d$ in stage 3 and use it as a parameter to adjust latency.

The IMC block computes $\mathbf{H}_b = \mathbf{J}_b^T\mathbf{J}_b$. We exploit the sparsity of $\mathbf{J}_b$ and block it, by combining identity matrices and null matrices, in the form shown in Fig. 6. This blocking reduces the computation $\mathbf{J}_{b_{30\times15}}^T\mathbf{J}_{b_{15\times30}}$ to $\mathbf{\Lambda}_{24\times9}^T\mathbf{\Lambda}_{9\times24}$. The required number of multiply-accumulations is reduced by 61.6%. The computations are also mapped onto multiple MAC units.

3.5 Storage Optimization of S Matrix

After the SE block and the IMC block compute the corresponding elements of $\mathbf{S}$ matrix, $\mathbf{S}$ matrix is constructed and stored in on-chip memory. Direct storage of $\mathbf{S}$ which has $(15n)^2$ elements occupies large on-chip memory space, contributing about 60% of the total on-chip memory requirement. We optimize $\mathbf{S}$ storage by separately storing the visual contribution $\mathbf{S}_c$ and the IMU contribution $\mathbf{S}_b$ and exploiting their individual symmetry with structured sparsities.

Fig. 9a illustrates the example of storing visual contribution $\mathbf{S}_c$ with $n = 5$ keyframes. Note that visual information only provides translation p and rotation q of pose s. Therefore, the non-zero elements of $\mathbf{S}_c$ exist only in a 6×6 subblock. Exploiting the symmetry can further compact about a half space.

Fig. 9b demonstrates the storage of $\mathbf{S}_b$. Because an IMU observation is related only to the adjacent keyframes, the non-zero elements of $\mathbf{S}_b$ exist only in the diagonal and sub/super-diagonal blocks (Step1). Then by exploiting symmetry the storage is reduced about a half (Step2). Finally, leveraging fine-grained sparsity existing in the remaining

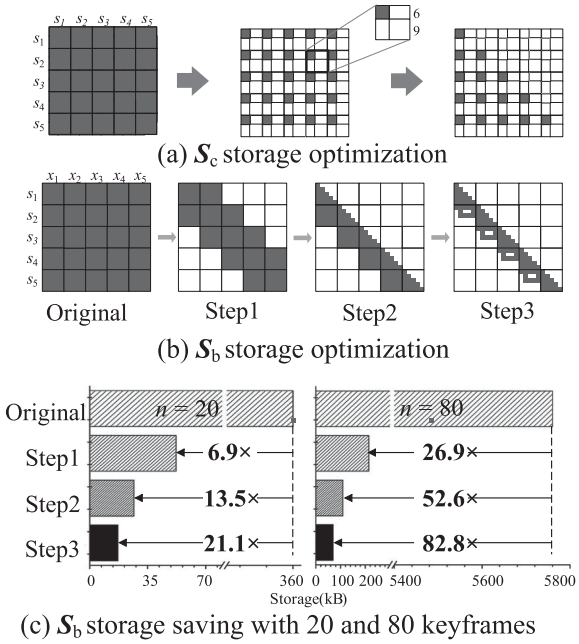


Fig. 9. Matrix S storage optimization.

matrix can further reduce the storage (Step3). Assuming each element is 32 bits, the storage optimization for $\mathbf{S}_b$ saves 81.3% memory space in this example. As the number of keyframes increases, the saving is more significant, as shown in Fig. 9c with $n = 20$ and $n = 80$.

Overall, our optimization reduces the memory space requirement from $(15n)^2$ to $18n^2 + 252n - 99$, in which $\mathbf{S}_c$ requires $18(n^2 + n)$ and $\mathbf{S}_b$ requires $234n - 99$.

3.6 Cholesky Decomposition Block

The Cholesky decomposition (CD) block solves $\mathbf{S}\delta\mathbf{p}_y = \mathbf{r}$. While Cholesky decomposition is not unique to SLAM, we leverage SLAM-specific knowledge to optimize its hardware design.

Recall that Cholesky decomposition decomposes a symmetric matrix $\mathbf{S}$ into LL^T, where L is a lower triangular matrix. The hardware iteratively generates columns of L, starting from the first column. For simple explanation, assuming that $\mathbf{S}$ is of dimension $n_k \times n_k$, the first iteration calculates the first column of L (the Evaluate phase), using which a new matrix $\mathbf{S}_{n_k-1}$ of size $(n_k - 1) \times (n_k - 1)$ is generated (the Update phase). $\mathbf{S}_{n_k-1}$ becomes the input matrix to the second iteration. This process continues until the entire L is calculated.

Fig. 10 shows the hardware design of the CD block, which cascades the Cholesky process elements (CPEs), each containing the Evaluate and Update units and completing one iteration of the Cholesky decomposition. Analyzing the fine-grained data dependencies shows that the Evaluate phase and the Update phase could be pipelined [18]. However, the computation latency of the Update phase is longer than that of the Evaluate phase and is reduced as iterating.

To ensure a balanced pipeline, our design uses multiple CPEs, the exact number of which is parameterized $\mathbf{n}_s$. Fig. 11 shows the timeline of execution under 6 CPEs, where E_i and U_i denote the latency of the Evaluate and Update phases in iteration i, respectively. Every 6 Evaluate-Update

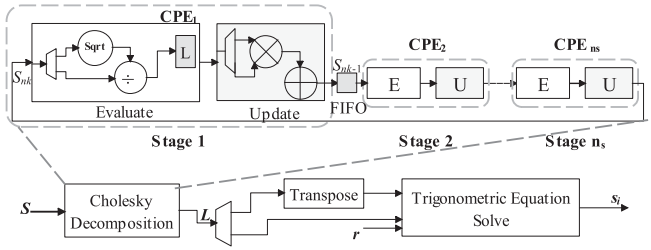


Fig. 10. The microarchitecture of the Cholesky decomposition block.

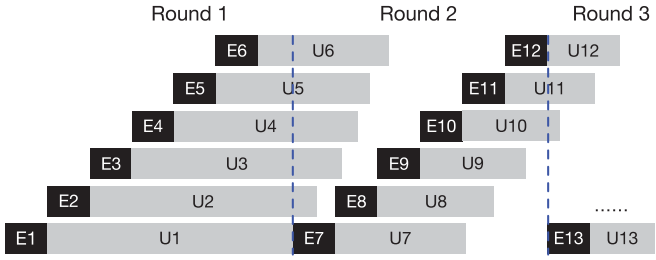


Fig. 11. The execution timeline of the CD block. E_i and U_i denote the Evaluate and Update latency in iteration i, respectively. Three rounds of execution are shown, each round finishes 6 Evaluate and Update phases, where 6 is the number of CPEs.

iterations form a round; the next round could only start when the Evaluate and Update units are both available. As shown in Fig. 11, there are two cases: 1) when an Update unit is available later than the Evaluate unit (e.g., Round 1), and 2) when the Evaluate unit is available later than an Update unit (e.g., Round 2).

3.7 Marginalization Block

The marginalization (MAR) block computes $\mathbf{H}_p$ and $\mathbf{r}_p$ as described in Section 2.2. The computation of $\mathbf{H}_p = \mathbf{A} - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{\Lambda}^T$ is similar to $\mathbf{H}_c$. If two computations can share the same circuit, hardware resources are saved. However, the SE block cannot be directly used, because $\mathbf{M}$ is a generic matrix while $\mathbf{U}$ in calculating $\mathbf{H}_c$ is a diagonal matrix that can be trivially inverted. Calculating $\mathbf{M}^{-1}$ requires $O(m + 15)^3$ computation complexity.

To deal with the issue, we block $\mathbf{M}$ as $\begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix}$, where $\mathbf{M}_{11}$ contains the pose s information and $\mathbf{M}_{22}$ contains the feature point coordinate λ information. Then, $\mathbf{M}^{-1}$ can be expressed in the following form

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{M}_{11}^{-1} + \mathbf{M}_{11}^{-1}\mathbf{M}_{12}\mathbf{S}'^{-1}\mathbf{M}_{21}\mathbf{M}_{11}^{-1} & -\mathbf{M}_{11}^{-1}\mathbf{M}_{12}\mathbf{S}'^{-1} \\ -\mathbf{S}'^{-1}\mathbf{M}_{21}\mathbf{M}_{11}^{-1} & \mathbf{S}'^{-1} \end{bmatrix}, \tag{4}$$

where $\mathbf{S}' = \mathbf{M}_{11} - \mathbf{M}_{12}\mathbf{M}_{22}^{-1}\mathbf{M}_{21}$. Here, $\mathbf{M}_{22}$ is a diagonal matrix and thus the computation of $\mathbf{S}'$ can share the SE block. Then, the CD block is used to obtain L' of $\mathbf{S}'$, in order to derive $\mathbf{S}'^{-1}$.

Given L', the remaining computation is performed by the MAR block whose detailed structure is shown in Fig. 12. The main computation is matrix multiplications which are mapped onto the MAC units. Because $\mathbf{M}^{-1}$ is a dense matrix, the computation of $\mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{\Lambda}^T$ is time-consuming. Multiple MAC units are used to parallelize the computation to balance latency. Therefore, the MAR block is parameterized by the number of MAC units $\mathbf{n}_m$.

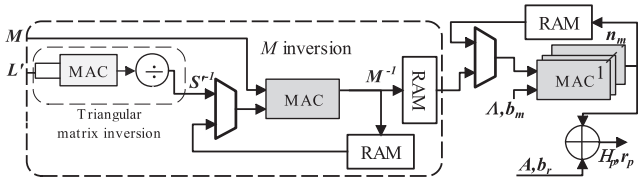


Fig. 12. The microarchitecture of the marginalization block.

4 Fixed Point Design

The backend of SLAM usually uses floating-point, even double precision, arithmetic to provide sufficient numerical precision [7]. However, the resource usage and consequently power consumption of hardware accelerator scales with the numeric precision of the computation. We identify the diverse features of data processed in SLAM, and use fixed-point (FXP) arithmetic to further improve efficiency of the accelerator. The FXP format for each variable involved in SLAM is determined with respect to the physical meanings, numerical dynamic range, the arithmetic computation and hardware resource cost.

We first present the fixed-point design flow here and then introduce the specific designs for each block in the following subsections. Signed two's complement fixed-point (FXP) numbers can be represented as $Q(QI, QF)$, in which QI is the bit-width of the integer part including the sign bit, and QF is the bit-width of the fractional part.

Because of significant impact on localization accuracy, QI is first determined according to the numerical range $[c_{min}, c_{max}]$ of each variable c involved in each block as (5). The range of c can be profiled during execution of the SLAM software implementation

$$QI(c_{min}, c_{max}) = \lceil \log_2 \max(|c_{min}|, |c_{max}|) + 1 \rceil \tag{5}$$

Then QF is chosen with respect to the trade-off between resource usage and precision requirement. The process is the following. In the first step, QI is selected on the base of computational resource. Table 1 lists the resource usage of FXP multiplier with different input bit-widths from Xilinx IP cores. We use the number of DSP blocks (#DSP) as the metric because DSP is the resource bottleneck for the accelerator design. For each variable with the determined QI, the closest input bit-width from the table is selected. For example, if c with $QI_c = 10$ is an operand to a multiplier, 18-bit bit-width is selected, i.e., $QF_c = 8$. If the bit-width of another operand is not greater than 25-bit, one DSP is used. The initial QF of all variables are selected following the same way.

In the second step, the Relative Pose Error (RPE), which is used to represent the location accuracy, is evaluated. If the RPE meets the requirement, then the FXP design is done; otherwise moving to the next step.

In the third step, the variables which significantly affect the accuracy are searched, such as those which suffer from severe underflows. For these variables, their bit-width is increased to the next level in Table 1 by increasing the fraction bit-width. Continuing with the previous example, if c is the one whose precision significantly affects RPE, the bit-width of c takes 25 or 35 bits and two DSPs are used. Afterwards, the process goes to the second step again.

TABLE 1
Resource Usage of FXP Multiplier

Bit-width	#DSP
18×25	1
25×35	2
40×35	4

In addition, the different blocks show various characteristics which also affect the FXP design. Next, FXP designs of the key blocks are presented.

4.1 Visual and IMU Jacobian

According to the specific mathematical properties and physical meaning of the data involved in the calculation of the visual and IMU Jacobian blocks, we set an appropriate fixed-point bit-width for them.

As mentioned, the data calculated in the visual block mainly include rotation matrix, coordinates, visual reprojection residual and Jacobian matrix. The rotation matrix satisfies the properties of the orthogonal matrix (the value of each element belongs to [-1,1]), and the continuous product between the rotation matrices is still a rotation matrix. Therefore, QI for the rotation matrix is set to 2.

The coordinate values in various coordinate systems have different units and have a wide range from several meters to several hundred pixels, so their QI is set to 10. The calculated residuals are in the unit of pixel, which are in the range of tens, and thus their QI is set to 8.

The element of Jacobian matrix is usually large, up to several thousands, so we set their QI to 15. Since there are division operations in the Jacobian matrix calculation process, it is necessary to first determine whether the divisor is 0 due to the FXP quantization. If the devisor is 0, its LSB is set to 1 to improve the calculation accuracy as much as possible. The overflow mode of the FXP design is set to saturation.

The precision of residual and Jacobian has significant impact on the location accuracy, and thus 25-bit and 35-bit bit-widths are chosen to determine QF. The FXP format of the data involved in the calculation of the IMU Jacobian block can be determined similarly. Table 2 shows FXP format of each variable of the visual and IMU Jacobian blocks.

4.2 Other Blocks

Following the similar method, the FXP format of the other blocks is determined as in Table 2. The orders of magnitude of data in the CD block vary significantly from 10^{-4} to 10^{3}. Direct FXP quantization will require large bit-width or cause a large loss of precision. Therefore, the data are normalized first into (-1, 1). The selection of the normalization scaling factor α ensures that the multiplication and division between the data and the factor can be obtained by shifting without overflow. The decomposition with the normalization becomes $\frac{\mathbf{S}}{\alpha} = \frac{L}{\sqrt{\alpha}}(\frac{L}{\sqrt{\alpha}})^T$. After normalization, QI of the data is set to 1 and QF is 17 or 24.

Because the SE and IMC blocks perform calculation with the Jacobian matrix, the FXP format of the involved variables are set as the same to that of the Jacobian matrix.

TABLE 2
FXP Format of Main Variables of the Different Blocks

Visual and IMU blocks	FXP format	Other blocks	FXP format
rotation matrix	(2, 23)	CD	(1, 17/24)
coordinate	(10, 25)	SE	(15, 20)
observation	(10, 25)	IMC	(15, 20)
velocity	(8, 27)	MAR	(15, 10/20)
translation	(8, 27)		
bias	(4, 21)		
residual	(8, 27)		
Jacobian matrix	(15, 20)		

The inputs of the MAR block are from the SE, IMC and CD blocks, the Q format of the data calculated in the MAR block is set as (15, 10) or (15, 20).

Section 6 will show that the above FXP designs lead to resource saving, speed improvement and power consumption reduction while meeting the location accuracy requirement.

5 Surrounding-Aware Runtime Configuration

As discussed in [12], the amount of workloads required in different driving environments often varies over time in order to sustain the accuracy in a real-world SLAM deployment. If a sliding window has fewer feature points, the accuracy would drop because less information is available to estimate the pose. In this case, the number of iterations in the iterative NLS solver is increased to sustain the accuracy level. More iterations lead to more work and, thus, require more powerful hardware configurations within the latency constraint.

A static hardware configuration is necessarily conservative to accommodate a large iteration count. This would degrade the efficiency of the accelerator when entering a new environment with sufficient feature points. Therefore, the proposed accelerator architecture is configured at runtime to adapt to different surroundings. The customized architecture presented in the previous section allows us to reconfigure the accelerator by adjusting the parameters of the three configurable blocks: the number of MAC units $\mathbf{n}_d$ in the SE block, the number of CPEs $\mathbf{n}_s$ in the CD block, and the number of MAC units $\mathbf{n}_m$ in the MAR block, respectively.

5.1 Problem Formulation

The determination of the three parameters can be formulated in the form of a constrained optimization problem, which minimizes system power consumption $Power(\cdot)$ with respect to latency $Lat(\cdot)$ and resource $Res(\cdot)$ constraints

$$\min_{\mathbf{n}_d,\mathbf{n}_m,\mathbf{n}_s} \quad Power(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s)$$
$$s.t.\ Lat(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) \leq L^*,\ Res(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) \leq R^*, \qquad (6)$$

where $Power(\cdot)$, $Lat(\cdot)$, and $Res(\cdot)$ are functions of $\mathbf{n}_d$, $\mathbf{n}_m$, and $\mathbf{n}_s$. L^* is the latency constraint specified by the designer, and R^* is the resource constraint imposed by a particular FPGA device. In addition to power, speed and resource, it is also possible to add other design metrics to the formulation,

such as localization accuracy. The formulation actually models the design space of accelerator for robot localization and can be used for design exploration at the design stage. In this work, the formulation is solved for runtime configuration of the accelerator. Next, we will show how the latency, power consumption and resource usage models are analytically formulated, respectively. We will omit the detailed derivation of the models due to space limit.

5.2 Analytical Modeling

Latency Model. The latency model is derived by calculating the critical path latency of the accelerator given the analytical latency models of each block

$$Lat(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) = Iter \times L_{NLS}(\mathbf{n}_d, \mathbf{n}_s) + L_{Marg}(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) \quad (7)$$

where L_{NLS} denotes the latency of an iteration of the NLS solver, $Iter$ denotes the total number of iterations in the NLS solver which varies with respect to the driving surroundings, and L_{Marg} denotes the marginalization latency.

The critical-path latency of an NLS iteration (the blocks along the solid lines in Fig. 2) is expressed as follows:

$$L_{NLS}(\mathbf{n}_d, \mathbf{n}_s) = \sum_{i=1}^{m} \max\{L_{VJac}, L_{Schur}(\mathbf{n}_d)\} + L_{Cholesky}(\mathbf{n}_s) + L_{Sub} \quad (8)$$

where L_{VJac} denotes the latency of the Visual Jacobian matrix computation, L_{Schur} denotes the latency of Schur elimination and is parameterized by $\mathbf{n}_d$, $L_{Cholesky}$ is the latency of Cholesky decomposition and is parameterized by $\mathbf{n}_s$, and L_{sub} denotes the back substitution latency with a fixed value independent of $\mathbf{n}_d$ and $\mathbf{n}_s$. The max operation reflects the pipeline between calculating the visual Jacobian and the Schur elimination across all m feature points. The amount of computations in the IMU Jacobian block is much less than the visual Jacobian block, so the IMU Jacobian and IMC blocks are not the critical path.

Given the statistically-balanced pipeline design in Section 3.2, the total latency L_{VJac} of calculating the Jacobian matrix elements for a feature point, ignoring the pipeline initiation, is given by

$$L_{VJac} = N_o C_o. \quad (9)$$

In the SE block, given the number of MAC units $\mathbf{n}_d$, the latency of performing Schur elimination for a feature point is

$$L_{Schur}(\mathbf{n}_d) = (6N_o)^2/\mathbf{n}_d \quad (10)$$

For the CD block, given $\mathbf{n}_s$ CPEs, the $n_k \times n_k$ input matrix $\mathbf{S}$, and the latency of the Evaluate unit L_e, the total latency is

$$L_{Cholesky}(\mathbf{n}_s) = \sum_{i=0}^{\lfloor \frac{n_k}{\mathbf{n}_s} \rfloor} max\{\mathbf{n}_s L_e, Le + \frac{m_i(m_i-1)}{2}\}, \quad (11)$$

$$m_i = n_k - i\mathbf{n}_s - 1 \quad (12)$$

where the max operation models the two scenarios of availability of the Evaluate and Update units.

The marginalization latency is the cumulative latency of the visual Jacobian, SE, CD and MAR blocks, which are along the dash lines in Fig. 2

$$L_{Marg}(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) = a_m L_{VJac} + L_{Schur}(\mathbf{n}_d) + L_{Cholesky}(\mathbf{n}_s) + L_{Mar}(\mathbf{n}_m) \quad (13)$$

Given $\mathbf{n}_m$ MAC units, the latency of the MAR block is

$$L_{Mar}(\mathbf{n}_m) = 15a_m + a_m^2 + b_k(15 + a_m)(6b + 9) + b_k(6b+9)^2 + 1250, \qquad b_k = \frac{15 + a_m}{\mathbf{n}_m} \quad (14)$$

where a_m denotes the number of feature points that are moved out of the current sliding window, and b denotes the number of keyframes in the sliding window.

Resource Model. The resource consumption of the accelerator is modeled as

$$Res(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) = R_0 + \mathbf{n}_d \times R_d + \mathbf{n}_m \times R_m + \mathbf{n}_s \times R_s \quad (15)$$

where R_d, R_m and R_s denote the *unit* resources consumption of the hardware structures that are parameters by $\mathbf{n}_d$, $\mathbf{n}_m$, and $\mathbf{n}_s$, respectively, and R_0 denotes the resource consumption independent of customization. For FPGA-based accelerator design, four major types of resources are considered, including lookup table (LUT), flip-flop (FF), on-chip RAM block (BRAM) and digital signal processing block (DSP). The resource constraint must be met for all four resource types.

Power Model. We model the power consumption similarly

$$Power(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s) = P_0 + \mathbf{n}_d \times P_d + \mathbf{n}_m \times P_m + \mathbf{n}_s \times P_s. \quad (16)$$

The coefficients P_d, P_m, and P_s are fitted for a particular FPGA device using regression models offline. This strategy adapts to other FPGAs and does not require measuring the power of individual blocks on an FPGA fabric, which is difficult to obtain in practice.

The formulation in (6) is a 3-variable mixed-integer convex programming problem. A near-optimal solution can be found in milliseconds using optimization packages such as YALMIP [19] and MLOPT [20].

5.3 Runtime Configuration Scheme

We design a low-overhead scheme to dynamically adjust the hardware architecture by building relationships among the surroundings (represented by the number of feature points), the NLS iteration count $Iter$ and the parameters $(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s)$.

A lookup table that maps the number of feature points to $Iter$ is first built by profiling datasets of interest offline [21]. In practice, we find that the NLS with $Iter > 6$ usually provides little accuracy improvement. Therefore, the maximum of $Iter$ is set to 6 and $Iter$ is adjusted during runtime, which is similar to the concept of approximate execution in autonomous

control [22]. The approximation introduces acceptable accuracy loss, which will be evaluated in Section 6.

Given a different $Iter$, the problem (6) needs to be solved again to generate a set of $(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s)$. Since there are only 6 $Iter$ choices, (6) is solved exhaustively against all $Iter$ values offline. This builds the second lookup table for mapping $Iter$ to $(\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s)$.

At runtime, the sensing frontend provides the number of feature points. Then, $Iter$ is adjusted when the number of feature points maps to a different $Iter$ in two consecutive sliding windows. Finally, the $\mathbf{n_d}$, $\mathbf{n_m}$, and $\mathbf{n}_s$ values are looked up and passed to the FPGA if different from the current values. In this way, runtime reconfiguration effectively has no overhead.

The FPGA accelerator, after obtaining the three parameters $\mathbf{n}_d, \mathbf{n}_m, \mathbf{n}_s$, uses the clock gating logic to only switch on corresponding computing units. In this way, the accelerator's power consumption is dynamically managed when entering different environments. Note that the runtime configuration does not require FPGA reconfiguration. Instead, the runtime configuration only involves clock gating scheme changes, i.e., some computing units' clock is off and some is on. The transition time is negligible by controlling the enable signals of the FPGA build-in clock buffers [23].

6 Experimental Results

To evaluate the proposed accelerator, we carry out a set of experiments. We first introduce the experimental setup (Section 6.1). Then the experiment results are presented in five parts (Section 6.2). The first part evaluates the speed and energy of the accelerator by comparing to software implementations and a hardware accelerator. The second part shows that the fixed-point design effectively reduces resource usage compared to floating-point arithmetic. The third part shows that the accelerator achieves satisfactory localization accuracy. The fourth part demonstrates that the accelerator is capable of runtime configuration to reduce power consumption. Finally, the accelerator is compared to several existing SLAM accelerators to demonstrate the design superiority.

6.1 Experimental Setup

The accelerator is designed with FXP numbers using Verilog, and synthesized and implemented onto the Xilinx Zynq-7000 SoC ZC706 FPGA using Vivado Design Suite 2018.2. The accelerator operates at a fixed frequency of 154 MHz. The power consumption of the accelerator is estimated using the Vivado power analysis tool using real workloads under test. All power and resource consumption data are obtained after the design passes the post-layout timing analysis.

The accelerator design is evaluated using two common datasets: KITTI Odometry [24] (grayscale sequence) for self-driving cars and EuRoC [25] (Machine Hall sequences) for drones. We select eight traces from the datasets with different number of keyframes (**#Keyframes**), feature points (**#Features**), visual observations (**#Visual obs**) and IMU observations (**#IMU obs**), representing eight driving scenarios.

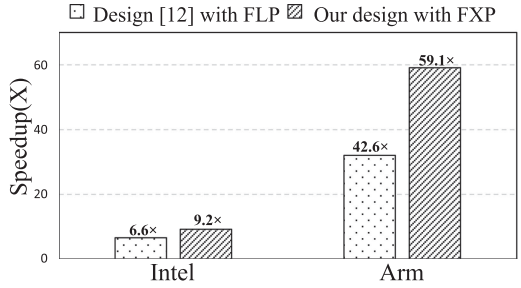


Fig. 13. Speed comparison to the software implementations and an FPGA implementation [12].

VINS-Fusion [26], a software implementation of SLAM, is used as a baseline, which uses Google's ceres solver [27] to implement bundle adjustment. The software implementation with double-precision floating-point (FLP64) numbers is parallelized through multithreaded vectorized CPU execution. The software is evaluated on two hardware platforms: one on an Intel Comet Lake processor that has 12 cores and operates at 2.9 GHz, and the other on the quad-core Arm Cortex-A57 processor on the Nvidia mobile Jetson TX1 platform [28] operating at 1.9 GHz. The Comet Lake's power is measured through a power meter and the Arm core power is measured through the power sensing circuitry on TX1.

6.2 Experimental Results

Performance Evaluation. We first evaluate the accelerator's performance and energy efficiency. The proposed accelerator with FXP is compared to the Intel CPU implementation, the Arm implementation and the FPGA implementation [12] with the single-precision floating-point (FLP) numbers.

Fig. 13 shows the result. The best speed of each implementation is measured and the speedup is averaged over the eight traces. Our accelerator, determined by transforming the problem (6) to minimize latency with respect to FPGA resource constraints, achieves $9.2\times$, $59.1\times$ and $1.4\times$ speedup over the two software implementations and the FLP FPGA implementation, respectively. The speed advantage of our design over [12] is from two aspects. One is that the FXP computation latency is shorter than the FLP computation. Another is that the FXP computation consumes hardware resources less than the FLP computation and thus our design has higher parallelism, i.e., larger $\mathbf{n}_d$, $\mathbf{n}_m$ and $\mathbf{n}_s$. For example, when testing on trace 1, the FLP implementation has $\mathbf{n}_d = 10$, $\mathbf{n}_m = 56$ and $\mathbf{n}_s = 97$, while our FXP implementation has $\mathbf{n}_d = 10$, $\mathbf{n}_m = 113$ and $\mathbf{n}_s = 227$.

Fig. 14 demonstrates the energy efficiency. We randomly set two latency constraints, 20ms and 33ms, corresponding to two scenarios. The short latency constraint requires fast localization response, emulating the high speed driving. The power-optimized accelerator designs are generated from the solving the problem (6). Our design achieves $153\times$, $33\times$ and $2\times$ energy reduction under the 20ms constraint; $104\times$, $22\times$ and $1.5\times$ energy reduction under the 33ms constraint, over the Intel, Arm and FPGA accelerator [12], respectively. Our design demonstrates superior energy efficiency under the tight constraint. Again, compared to the FLP design [12], our design achieves better energy efficiency.

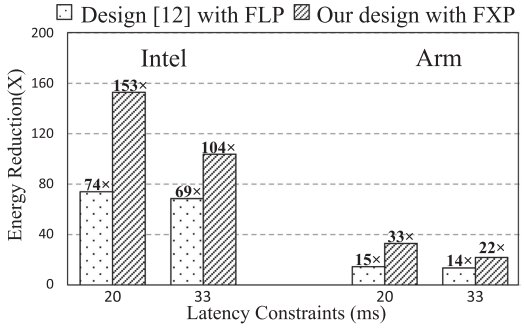


Fig. 14. Energy comparison to the software implementations and an FPGA implementation [12] under two latency constraints.

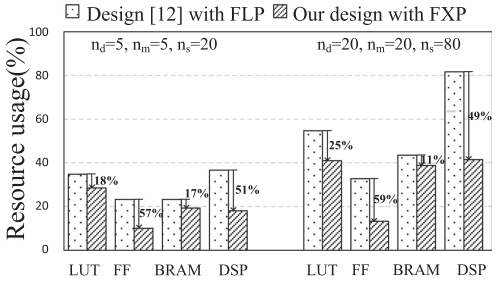


Fig. 15. Resource usage comparison to the FPGA implementation [12] under two architecture configurations.

Resource Usage Evaluation. Next, let us closely look at the resource usage of the proposed accelerator. Fig. 15 shows utilization percentages of FPGA computation and memory resources of our accelerator and the FPGA implementation [12]. Two architecture configurations are selected, representing the low and high parallelism, respectively. At the low parallelism configuration ($\mathbf{n}_d = 5$, $\mathbf{n}_m = 5$, $\mathbf{n}_s = 20$), our FXP accelerator saves 18% LUTs, 57% FFs, 17% BRAMs and 51% DSPs. At the high parallelism configuration ($\mathbf{n}_d = 20$, $\mathbf{n}_m = 20$, $\mathbf{n}_s = 80$), our FXP accelerator saves 25% LUTs, 59% FFs, 11% BRAMs and 49% DSPs. The DSP usage is significantly reduced, due to the FXP design strategy proposed in Section 4.

Localization Accuracy. In addition to performance and resource usage, we also care about the localization accuracy of the proposed accelerator. An open-source tool evo [29] is used to evaluate localization accuracy. It calculates two RPEs, Root Mean Square Error (RMSE) and Maximum Error (ME), between the location outputs of odometry and SLAM algorithms and the ground truth locations. The traces shown in Table 3 are used in this evaluation. For example, on the KITTI dataset, RMSE and ME are the average error and the maximum error calculated over the four traces (Num 5-Num 8), respectively. Fig. 16 illustrates the result. The baseline is the software implementation VINS-Fusion [26] with FLP64. An accelerator Archytas [12] is implemented with FLP32. The proposed accelerator with the FXP format is implemented with the static configuration (*Iter*=6) and the dynamic configuration. From the figure, we can observe the following results. First, the FXP hardware implementations achieve the similar localization accuracy. On the KITTI dataset, they increase RMSE and ME by upto 0.8cm and 1.9cm, respectively, to the baseline. On the EuRoC dataset, they increase RMSE and ME by upto 1.0cm and 2.5cm, respectively, to the baseline. The small accuracy loss is mainly caused by the FXP format. We can also observe accuracy loss of the FLP32 implementation to the baseline.

TABLE 3
Test Traces From the KITTI and EuRoC Datasets

Num	Trace	#Keyframes	#Features	#Visual obs	#IMU obs
1	MH01	10	148	1038	10
2	MH01	20	187	2688	20
3	MH01	30	210	4057	30
4	MH01	40	312	5067	40
5	KITTI00	30	417	3889	30
6	KITTI00	50	607	5950	50
7	KITTI01	30	269	2974	30
8	KITTI01	50	409	4414	50

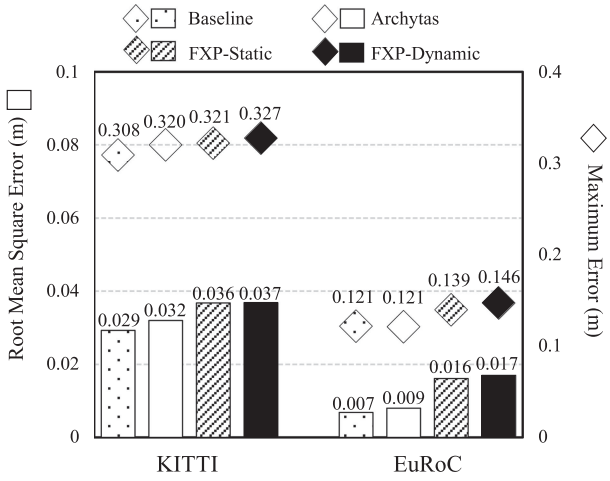


Fig. 16. Localization accuracy comparison. Baseline is the software implementation VINS-Fusion [26] with FLP64, and Archytas [12] is an accelerator with FLP32.

Second, compared to the static configuration, the dynamic configuration only increases the error by 2.2% on average, due to the dynamic adjustment of *Iter*.

To see how such RPEs affect the moving trajectory, we also plot the trajectories on two traces, Num 3 and Num 5 from Table 3, in Fig. 17. The FXP implementations generate the trajectories close to that of the FLP implementation and the ground truth. This is because the MAP estimation is robust in long-term localization, where the localization estimation error in a feature point could be complemented in the subsequent windows when more feature points are obtained.

Runtime Configuration Evaluation. We select 1000 consecutive sliding windows from the third trace in Table 3 to evaluate the runtime configuration performance of the designed accelerator. The 1000 sliding windows in sequence represent a driving path with various environments, where the number of feature points in each window varies over time. Again, we set two latency constraints: 33ms for the first 500 windows and 20ms for the second 500 windows.

Fig. 18 shows the power changes, over time, of the accelerator with the static configuration and the dynamic configuration. As shown in the figure, the static configuration considering the worst-case scenario consumes the peak power all the time. In contrast, the runtime configuration dynamically adjusts the architecture to adapt to different scenarios and saves power. During the second 500 windows, the power consumption changes more frequently due to the tight latency constraint and the dynamic scenarios. On average over the first 500 windows,

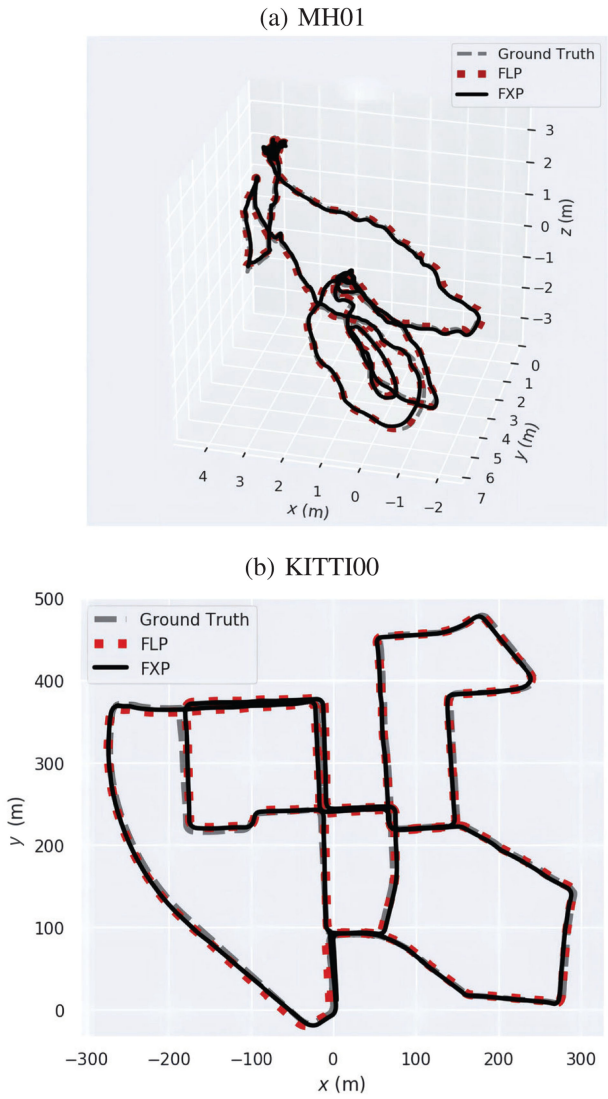


Fig. 17. Moving trajectory comparisons on a) EuRoC and b) KITTI datasets.

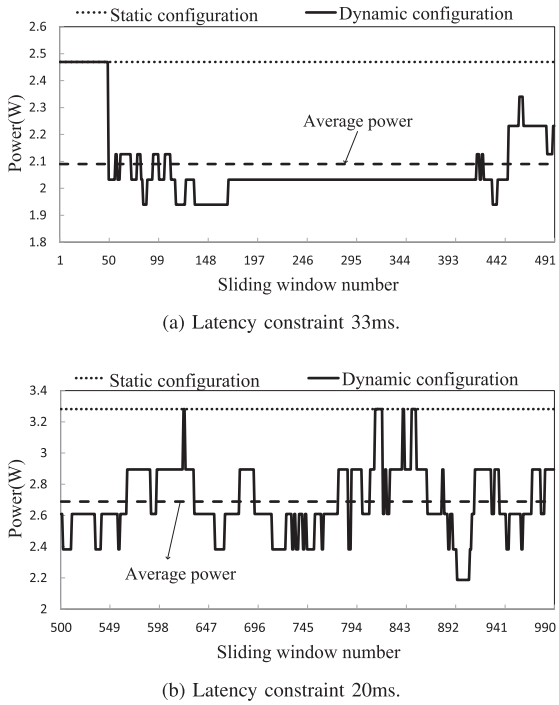


Fig. 18. Power consumption variation in the changing driving surroundings with the static and runtime dynamic configurations under two latency constraints.

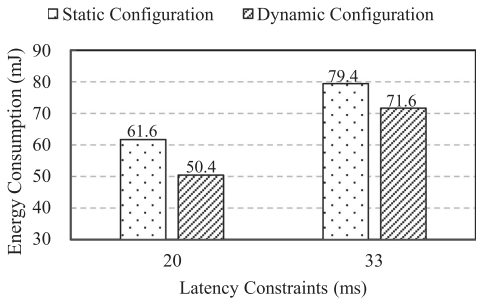


Fig. 19. Energy consumption comparison.

the runtime configuration saves power consumption 16%; over the second 500 windows saves 18%.

Note that the dynamic configuration reduces power consumption by reducing the parallelism of the computation and thus increases the processing latency, which still meets the latency constraint, compared to the static configuration. Therefore, we also compare the total energy consumption of the static and dynamic configurations, as shown in Fig. 19. The result shows that the dynamic configuration reduces the energy consumption by 18.2% and 9.8% under two latency constraints, respectively.

Comparison to Exiting SLAM Accelerators. Different localization accelerators target slightly different SLAM algorithmic variants/hardware platforms and are evaluated on different benchmarks, so a fair comparison is difficult. Table 4 is our best-effort comparisons with prior localization accelerators, where all evaluations focus on the backend part of SLAM.

From the table we can make the following observations. First, the ASIC implementation [7] has the best performance as expected. However, the ASIC designs face issues of high non-recurring engineering cost, long time-to-market and being not adaptive to localization algorithm evolution. Second, our design has the best performance and energy efficiency compared to the other FPGA implementations. The main gain comes from the SLAM-customized architecture design, the fixed-point design and the runtime configuration. Compared to [9] and [12], the performance improvement of our design mainly comes from the fixed-point design and the microarchitecture optimization such as the visual Jacobian block and the Schur elimination block. In addition, [9] only implements the Jacobian and Schur elimination in hardware and the rest in software. The backend of [7] and [5] uses Gauss-Newton algorithm and double floating-point numbers, while our design uses LM algorithm and fixed-point design leading to high efficiency. Similar to [9], Eudoxus [11] also uses hardware and software co-design, while our design implements the whole backend in hardware avoiding the communication cost between software and hardware. Third, our accelerator adapts to run-time dynamism while all prior accelerators do not.

TABLE 4
Comparison With Recent Backend Accelerators of SLAM

	This work	Archytas[12]	Navison[7]	pi-BA[9]	VIO on Chip[5]	Eudoxus[11]
Platform	FPGA	FPGA	ASIC	FPGA	FPGA	FPGA
Technology	28 nm	28 nm	65 nm	28 nm	28 nm	16 nm
Algorithm	LM-opt	LM-opt	GN-opt	LM-opt	GN-opt	Kal-filer
Sensor	Visual+IMU	Visual+IMU	Visual+IMU	Visual	Visual+IMU	Visual+IMU
Data Types	FXP	FLP32	FLP64+FXP	FLP32	FXP	FLP32
Voltage	1 V	1 V	1.2 V	1 V	1 V	0.85 V
Power	2.39 W	3.45 W	9.6 mW	5.50 W	612 mW	3.1 W
Frequency	154 MHz	143 MHz	62.5/83.3 MHz	143 MHz	100 MHz	180 MHz
Throughput	50 fps	50 fps	32 fps	4fps	5 fps	22 fps
Energy per Frame	35.6 mJ	56.6 mJ	300.0 uJ	605 mJ	122 mJ	141.0 mJ
Runtime Arch Configuration	Yes	Yes	N/A	No	No	No

All performance data of the compared accelerators are from the published papers.

7 Related Work

As more and more autonomous machines are deployed in real-world applications, various robotics tasks are accelerated for the real-time requirements on the mobile systems [7], [8], [9], [10], [11], [22]. This paper specifically focuses on localization.

Hardware acceleration of robotic localization is an active area of research. Prior accelerators target both ASIC [7], [8] and FPGA [5], [9], [10], [11], [30], [31], [32], and accelerate different algorithms including optimization-based algorithms [15] (e.g., BA) and filter-based algorithms [11]. This paper focuses on BA-based algorithms due to their wide applicability, and targets FPGA due to its design flexibility and rich sensor interfaces that ease its integration into an end-to-end computing system [33].

Some of the prior efforts focus on accelerating process of the visual SLAM. An energy-efficient architecture is proposed for real-time ORB-based visual SLAM system by accelerating the most time-consuming stages of feature extraction and matching on FPGA platform [32]. π-BA [9] accelerates BA, in which the Jacobian calculation and Schur elimination are implemented on FPGA and the rest is implemented on software. BAX [34] is a full hardware accelerator for BA, and uses generic vector units for acceleration. Gautier et al. [31] propose FPGA accelerator for a dense SLAM framework which performs 3D reconstruction based on an input from a RGB-D sensor. The target hardware platforms are low-power FPGA SoC and high-performance CPU-FPGA heterogeneous system. Similarly, Boikos et al. [30] accelerate the semi-dense LSD-SLAM algorithm on an FPGA SoC achieving 22 frames per second on a 320x240 input visual frame.

Visual-inertial odometry (VIO) SLAM uses camera and inertial measurement unit (IMU) data to estimate location and map the surroundings. Several FPGA accelerators target the VIO-SLAM by hardware-software co-design. Zhang et al. [5] jointly explore the algorithm and hardware design space for accelerating VIO-SLAM. The accelerator can adapt to environments by rescheduling the frontend and the backend pipelines to save time. Similarly, a framework for autonomous machine localization is proposed, which adapts to different operating scenarios by fusing fundamental algorithmic primitives and co-designing a hardware accelerator [11]. In [10], the sparsity feature of SLAM is deployed to accelerate EKF and ORB SLAM while co-optimizing power and latency.

The proposed accelerator differs from previous accelerators in two main ways. First, unlike most of the prior accelerators, which are designed for a particular design point, our accelerator with configurable architecture provides a hardware template, on top which a synthesis framework such as [12] can generate an accelerator given the design specifications. Second, our accelerator re-configures architecture at runtime to adapt to the driving environment, where in virtually all prior accelerators' architecture are static.

8 Conclusion

To address the challenges faced by accelerating localization, we propose an energy-efficient and runtime-configurable accelerator architecture. Leveraging SLAM-specific features, we design efficient computing structure with fixed-point arithmetic and reduce memory usage. Parameterized architecture allows us to 1) formulate the design space exploration into convex optimization problem, leading to automated and optimized design solution generation; and 2) adjust the accelerator at runtime to adapt to various driving environments and save power. Evaluations on FPGA and common datasets demonstrate that the proposed accelerator has superior performance and energy efficiency over the Intel CPU, Arm CPU and existing FPGA accelerators.

References

[1] S. Liu and J.-L. Gaudiot, "Rise of the autonomous machines," *Computer*, vol. 55, no. 1, pp. 64–73, 2022.
[2] C. Cadena et al., "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Trans. Robot.*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016.
[3] Z. Zhang, S. Liu, G. Tsai, H. Hu, C.-C. Chu, and F. Zheng, "PIRVS: An advanced visual-inertial Slam system with flexible sensor fusion and hardware co-design," in *Proc. Int. Conf. Robot. Automat.*, 2018, pp. 1–7.
[4] H. Strasdat, J. M. Montiel, and A. J. Davison, "Visual SLAM: Why filter?," *Image Vis. Comput.*, vol. 30, no. 2, pp. 65–77, 2012.
[5] Z. Zhang, A. A. Suleiman, L. Carlone, V. Sze, and S. Karaman, "Visual-inertial odometry on chip: An algorithm-and-hardware co-design approach," in *Proc. Robot. Sci. Syst.*, 2017, pp. 1–10.
[6] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[7] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A 2-mW fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones," *IEEE J. Solid-State Circuits*, vol. 54, no. 4, pp. 1106–1119, Apr. 2019.
[8] Z. Li et al., "An 879GOPS 243mW 80fps VGA fully visual CNN-SLAM processor for wide-range autonomous exploration," in *Proc. Int. Solid-Statist. Circuits Conf.*, 2019, pp. 134–136.
[9] Q. Liu, S. Qin, B. Yu, J. Tang, and S. Liu, "π-BA: Bundle adjustment hardware accelerator based on distribution of 3D-point observations," *IEEE Trans. Comput.*, vol. 69, no. 07, pp. 1083–1095, Jul. 2020.
[10] B. Asgari, R. Hadidi, N. Shoghi, and H. Kim, "PISCES: Power-aware implementation of SLAM by customizing efficient sparse algebra," in *Proc. Des. Automat. Conf.*, 2020, pp. 1–6.
[11] Y. Gan et al., "Eudoxus: Characterizing and accelerating localization in autonomous machines," in *Proc. Int. Symp. High Perform. Comput. Architect.*, 2021.
[12] W. Liu et al., "Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization," in *Proc. Int. Symp. Microarchitecture*, 2021, pp. 479–493.
[13] G. Sibley, "Sliding window filters for SLAM," University of Southern California, Center for Robotics and Embedded Systems, Tech. Rep. CRES-06-004, 2006.
[14] G. Sibley, L. Matthies, and G. Sukhatme, "Sliding window filter with application to planetary landing," *J. Field Robot.*, vol. 27, no. 5, pp. 587–608, 2010.
[15] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-manifold preintegration theory for fast and accurate visual-inertial navigation," *IEEE Trans. Robot.*, vol. 33, no. 1, pp. 1–21, Feb. 2017.
[16] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *J. Soc. Ind. Appl. Math.*, vol. 11, no. 2, pp. 431–441, 1963.
[17] F. Zhang, *The Schur Complement and its Applications*. vol. 4, Berlin, Germany: Springer Science & Business Media, 2006.
[18] J. Luo, Q. Huang, S. Chang, X. Song, and Y. Shang, "High throughput Cholesky decomposition based on FPGA," in *Proc. Int. Congr. Image Signal Process.*, 2013, pp. 1649–1653.
[19] J. Lofberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *Proc. Int. Conf. Robot. Automat.*, 2004, pp. 284–289.
[20] A. Cauligi, P. Culbertson, B. Stellato, D. Bertsimas, M. Schwager, and M. Pavone, "Learning mixed-integer convex optimization strategies for robot planning and control," in *Proc. Conf. Decis. Control*, 2020, pp. 1698–1705.
[21] M. Maimone, J. Biesiadecki, E. Tunstel, Y. Cheng, and C. Leger, "Surface navigation and mobility intelligence on the Mars exploration rovers," *Intell. Space Robot.*, pp. 45–69, 2006.
[22] B. Li, J. Tan, and K. Yan, "AERO: Towards energy-efficient autonomous flight in MAVs using approximate execution," in *Proc. IEEE 32nd Int. Conf. Appl.-Specific Syst., Architect. Processors*, 2021, pp. 196–202.
[23] S. Sundaresan and F. Rivoallon, "Analysis of power savings from intelligent clock gating," *XILINX, XAPP790 (v1. 0)*, vol. 13, 2012.
[24] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Conf. Comput. Vis. Pattern Recognit.*, 2012, pp. 3354–3361.
[25] M. Burri et al., "The EuRoC micro aerial vehicle datasets," *Int. J. Robot. Res.*, vol. 35, no. 10, pp. 1157–1163, 2016.
[26] "VINS-Fusion," 2021. [Online]. Available: https://github.com/HKUST-Aerial-Robotics/VINS-Fusion
[27] "Ceres Solver," 2021. [Online]. Available: http://ceres-solver.org/
[28] "TX1 datasheet," 2021. [Online]. Available: http://images.nvidia.com/content/tegra/embedded-systems/pdf/JTX1-Module-Product-sheet.pdf
[29] M. Grupp, "evo: Python package for the evaluation of odometry and Slam," 2017. [Online]. Available: https://github.com/MichaelGrupp/evo
[30] K. Boikos and C.-S. Bouganis, "Semi-dense SLAM on an FPGA SoC," in *Proc. Int. Conf. Field Programmable Log. Appl.*, 2016, pp. 1–4.
[31] Q. Gautier, A. Althoff, and R. Kastner, "FPGA architectures for real-time dense SLAM," in *Proc. Int. Conf. Appl.-Specific Syst., Architect. Processors*, 2019, pp. 83–90.
[32] R. Liu, J. Yang, Y. Chen, and W. Zhao, "eSLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform," in *Proc. Des. Automat. Conf.*, 2019, pp. 1–6.
[33] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, "Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations," in *Proc. Int. Symp. Microarchitecture*, 2020, pp. 1067–1081.
[34] R. Sun, P. Liu, J. Xue, S. Yang, J. Qian, and R. Ying, "BAX: A bundle adjustment accelerator with decoupled access/execute architecture for visual odometry," *IEEE Access*, vol. 8, pp. 75530–75542, 2020.

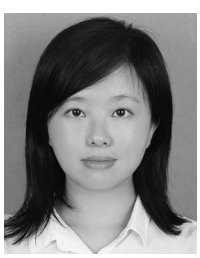
Qiang Liu (Member, IEEE) received the PhD degree from the Department of Electrical and Electronic Engineering at Imperial College London, London, U.K., in 2008. From 2009 to 2011, he was a research associate with the Department of Computing, Imperial College London. He is currently a professor with the School of Microelectronics, Tianjin University, with research interests in high speed and low power circuit system design, and reconfigurable computing.

Yuhui Hao received the BEng degree from the School of Microelectronics, Tianjin University, Tianjin, China, in 2021. He is currently working toward the ME degree with Tianjin University, Tianjin, China. His research interests include digital integrated circuit design and reconfigurable computing.

Weizhuang Liu received the BEng degree from the School of Microelectronics, Tianjin University, Tianjin, China, in 2019. He is currently working toward the ME degree with Tianjin University, Tianjin, China. His research interest is digital integrated circuit design.

Bo Yu (Senior Member, IEEE) received the PhD degree from the Institute of Microelectronics, Tsinghua University, Beijing, China, in 2013. His current research interests include algorithm and systems for robotics and autonomous vehicles. He is the founding member of the IEEE Special Technical Community on Autonomous Driving.

Yiming Gan received the bachelor's degree from the Beijing Institute of Technology, and the master's degree from the University of California, Santa Barbara. He is currently working toward the PhD degree with the Department of Computer Science, University of Rochester. He is now working on topics such as domain specific architecture for robust deep learning, reliable autonomous machines and robotic applications.



Jie Tang (Senior Member, IEEE) received the PhD degree from the Beijing Institute of Technology, in 2012. She is currently an associate professor with the School of Computer Science and Engineering of South China University of Technology, Guangzhou, China. She was previously a visiting researcher with the Embedded Systems Center at University of California, Irvine, USA, and a research scientist with Intel China Runtime Technology Lab. She is mainly doing research on computer architecture, autonomous driving and run-time system.

Shao-Shan Liu (Senior Member, IEEE) is leading the Autonomous Driving division as well as the Autonomous Mobile Clinics project with BeyonCa. His research focuses on autonomous driving technologies. He is a a member of AAAS and ACM.

Yuhao Zhu (Member, IEEE) is an Assistant Professor of computer science with the University of Rochester. His research group focuses on applications, algorithms, and computer systems for visual computing. His work is recognized by the Honorable Mention of the 2018 ACM SIGARCH/ IEEE-CS TCCA Outstanding Dissertation Award and multiple IEEE Micro Top Picks designations. He is a recipient of the NSF CAREER Award, in 2020.