Archytas: A Framework for Synthesizing and Dynamically Optimizing Accelerators for Robotic Localization

Weizhuang Liu*
Tianjing University
China
liuwz@tju.edu.cn

Bo Yu*
PerceptIn
USA
bo.yu@perceptin.io

Yiming Gan
University of Rochester
USA
ygan10@ur.rochester.edu

Qiang Liu
Tianjing University
China
qiangliu@tju.edu.cn

Jie Tang
SCUT
China
cstangjie@scut.edu.cn

Shaoshan Liu
PerceptIn
USA
shaoshan.liu@perceptin.io

Yuhao Zhu
University of Rochester
USA
yzhu@rochester.edu

ABSTRACT

Despite many recent efforts, accelerating robotic computing is still fundamentally challenging for two reasons. First, robotics software stack is extremely complicated. Manually designing an accelerator while meeting the latency, power, and resource specifications is unscalable. Second, the environment in which an autonomous machine operates constantly changes; a static accelerator design leads to wasteful computation.

This paper takes a first step in tackling these two challenges using localization as a case study. We describe Archytas, a framework that automatically generates a synthesizable accelerator from the high-level algorithm description while meeting design constraints. The accelerator continuously optimizes itself at run time according to the operating environment to save power while sustaining performance and accuracy. Archytas is able to generate FPGA-based accelerator designs that cover large a design space and achieve orders of magnitude performance improvement and/or energy savings compared to state-of-the-art baselines.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**; **Robotics**; *Reconfigurable computing*.

KEYWORDS

robotics, localization, SLAM, bundle adjustment, accelerator, FPGA, optimization, hardware synthesis, run-time system

*Both authors contributed equally to this research.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480077

ACM Reference Format:
Weizhuang Liu, Bo Yu, Yiming Gan, Qiang Liu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2021. Archytas: A Framework for Synthesizing and Dynamically Optimizing Accelerators for Robotic Localization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480077

1 INTRODUCTION

Robotic computing has reached a tipping point. A myriad of autonomous machines such as drones, logistic robots, and self-driving cars are on the cusp of becoming an integral part of our everyday life. The continuous proliferation of autonomous machines, however, depends critically on an efficient computing substrate, driven by higher performance requirements and the miniaturization of machine form factors. As a result, it comes with no surprise that a great amount of recent efforts focus on accelerating various robotics tasks [9, 24, 36, 39, 42, 44, 45, 59–61, 68, 73].

This paper specifically focuses on localization, i.e., calculating the position of an agent in the environment. Accelerating localization has two challenges. First, to arrive at a practical design for production, designers often must explore a large design space delineated by the performance requirement, resource constraints, and the expected accuracy level. However, the robotics software is often extremely complicated. For instance, the popular localization framework VINS [5, 66] has about a hundred thousand lines of code. Thus, manually exploring the design space to derive an optimal accelerator design is unscalable. A slight modification to the algorithm, target hardware platform, or performance requirement would potentially require a manual redesign.

Second, the environment that an autonomous machine operates in is constantly changing, and thus the load of work at run time is dynamically changing. A purely static accelerator, as in virtually all prior work, accommodates the worst case, necessarily leading to wasteful computation at run time.

This paper presents Archytas, a framework that automatically generates localization accelerators (in the form of synthesizable

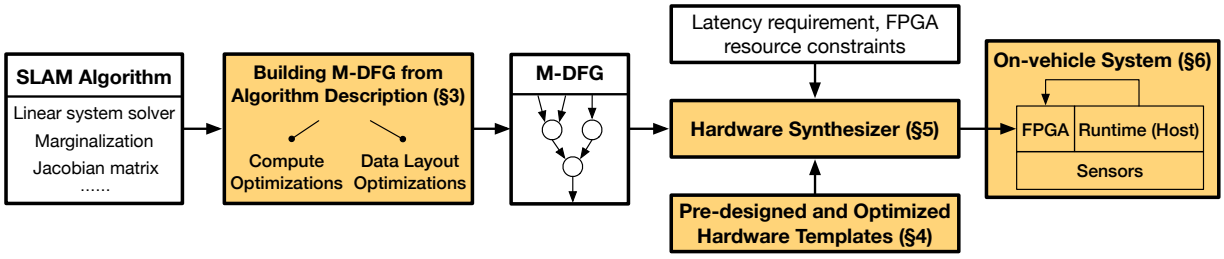


Fig. 1: System overview. Archytas first generates a M-DFG from a high-level algorithm description. The M-DFG is then mapped to a parameterized hardware template, which is concretized by the hardware synthesizer (in the form of synthesizable Verilog code) given the latency and power specifications. The run-time system continuously re-optimizes the hardware according to the operating environment to opportunistically save power.

Verilog) from high-level algorithm descriptions given the power, latency, and resource specifications. At run time, the accelerator continuously re-optimizes itself according to the operating environment to save power while sustaining performance and accuracy.

We target a specific family of localization algorithms based on maximum a posteriori (MAP) estimation [15]. The MAP formulation is widely used in mobile robots [67], aerial robots [5], autonomous vehicles [76], and Mars rovers [19]. MAP allows an autonomous machine to localize in previously-unseen environments [56] and to correct cumulative errors during long-term, continuous localization [72]; both are key to real-world deployment. Fig. 1 shows an overview of the Archytas system.

Static Synthesis In generating the accelerator statically, Archytas addresses two challenges. First, while the general localization algorithm description is well-established, many algorithmic blocks, such as matrix inversion and linear system solver, can be implemented in different ways. By exploiting domain-specific knowledge (e.g., inherent data sparsity, data-flow across blocks), Archytas generates a concrete software implementation of localization, in the form of a macro data-flow graph (M-DFG), which minimizes the computation cost for each algorithmic block and facilitates hardware resource sharing across blocks (Sec. 3).

Second, given the M-DFG, Archytas generates hardware using a pre-designed template. We show how to optimize the key hardware blocks in the template by exploiting the locality and parallelism unique to localization (Sec. 4). Critically, we identify a set of "customizable" hardware blocks whose resource provisions dictate the resource-vs-latency trade-off. Given the design specifications (e.g., resource and latency constraints), Archytas generates the exact configurations of the customizable blocks via constrained optimization, yielding a concrete accelerator (Sec. 5).

Dynamic Optimization At run time, Archytas dynamically re-optimizes its hardware configuration to minimize power (Sec. 6). We leverage the observation that the load of work at run time varies with the environment in which an autonomous machine operates. Archytas dynamically scales down the hardware provision when the workload decreases with little accuracy/latency impact. Critically, we show that almost all run-time decisions could be made offline and memoized, leading to negligible run-time overhead.

Results We generate designs targeting the Xilinx Zynq-7000 FPGA platform. Archytas is able to generate designs that cover a design space of 5× performance difference and 2× power difference. Evaluating on two common datasets, KITTI Odometry [26] for self-driving cars and EuRoC [14] for drones, an Archytas-generated accelerator is able to provide a 6.2× speedup and 74.0× energy reduction over a 16-core Intel Comet Lake CPU and a 39.7× speedup and 14.6× energy reduction over a quad-core Arm A57 core. The dynamic optimization introduces another 2× energy reduction without degrading the accuracy or performance.

In summary, this paper makes the following contributions:

- We observe that there exists a large software design space for the generic localization algorithm. We exploit localization-specific knowledge to generate a concrete software implementation (an M-DFG graph) from the algorithm description that minimizes the overall computation and enables hardware resource sharing.
- We present a template hardware for accelerating localization. We show how to optimize key hardware blocks in the template by exploiting the data sparsity, locality, and parallelism inherent in localization.
- We present a hardware synthesis framework that automatically generates a concrete localization accelerator given the M-DFG and the hardware template. We demonstrate that the generated accelerators can flexibly trade performance for energy and out-perform multi-core baselines and existing localization accelerators.
- We introduce a run-time system that dynamically re-optimizes the hardware to save power according to the operating environment without degrading accuracy and performance.

2 BACKGROUND

We first introduce the scope of the localization problem (Sec. 2.1), followed by describing the most widely used SLAM algorithm (Sec. 2.2), which we target in this paper.

2.1 Simultaneous Localization and Mapping

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) [15]. While other localization settings are possible (e.g., assuming the environment

map is available), this paper focuses on SLAM since it is the least restrictive and is widely used in virtually all autonomous machines, such as self-driving cars [24], Augmented Reality [1], and drones [9, 73]. SLAM is also central to many offline applications such as 3D reconstruction, for which perhaps one of the most notable examples is the Google Street View [37].

Formally, a 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 $< x, y, z >$ coordinates, and the three *rotational* DoF, i.e., the orientation about the three orthogonal axes (a.k.a., yaw, roll, and pitch).

SLAM Algorithms A principled mathematical approach to solving SLAM is Maximum a posteriori (MAP) estimation [15]. Comparing to the other popular class of SLAM algorithm based on non-linear filtering [27, 40, 41, 41, 74, 86], MAP is more robust in long-term localization and is more efficient, as quantified by accuracy per unit of computing time [72].

While MAP is an extremely important subset of localization algorithms, it is not the only algorithm, nor is it necessarily the best algorithm for every localization scenario [24]. The filtering-based algorithm has seen significant improvements recently with Multi-State Constraint Kalman Filter-based algorithms [57] such as OpenVINS [28] and MSCKF VIO [74]. Our goal is to use MAP as a case study to demonstrate the Archytas approach, which automatically generate accelerators given design specifications while adapting to run-time dynamisms.

2.2 MAP Estimation Formulation

MAP estimation localizes an agent from a sequence of sensor measurements. MAP can be thought of as a "real-time version" of the conventional Bundle Adjustment (BA) algorithm used in offline tasks such as Structure from Motion [77]. Compared to offline BA, MAP for robotics has two key differences [15]. 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 IMU+camera setup.

Second, only a fixed number of measurements, maintained through a sliding window, are used for MAP estimation to ensure real-time performance [70], essentially providing an *incremental* BA. The incremental strategy is critical for real-time performance. As the window slides, the oldest measurements are moved out of the window. These measurements, however, are not discarded entirely. Instead, they become the *prior* in the MAP estimate, through a process known as marginalization [71], 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 1) the sequence of the machine's 6 DoF poses (i.e., localization) and 2) the 3D coordinates of the points in space captured by the camera (i.e., mapping) in the current window:

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

where s_i represents the 6 DoF pose at i-th measurement, and λ_j represents the 3D coordinates of j-th observed point.

The crux of MAP is to solve a nonlinear least squares (NLS) optimization problem to estimate $\mathbf{p}$ [22]:

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

where N is the number of sensors; $\mathbf{o}_i$ is the actual measurement of i-th sensor ("ground truth"), $\mathcal{P}_i$ is a function that maps the (to-be-estimated) state vector $\mathbf{p}$ to 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 [66, 71], respectively, and $|\cdot|^2$ (L2 norm) quantifies the errors imposed by the priors.

3 MAPPING ALGORITHM DESCRIPTION TO MACRO DATA-FLOW GRAPH

We first present the general MAP algorithm (Sec. 3.1). We then describe how to generate a concrete software implementation (Sec. 3.2) with optimal minimal storage requirement by exploiting characteristics of SLAM-specific data structures (Sec. 3.3).

3.1 The General Algorithm Description

Fig. 2 shows the overall structure of the algorithm to solve the MAP problem, which consists of two phases: 1) a NLS solver that solves Equ. 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 sequentially.

NLS Solver Archytas targets the Levenberg-Marquardt (LM) algorithm [54] widely-used in commercial products ranging from robotics, AR/VR, and 3D reconstruction [3]. The LM algorithm belongs to the class of gradient 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 multiplication, transpose, and addition 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 [17]:

(1) Calculate the Jacobian matrix $\mathbf{J}$ and the residual $\mathbf{e}$ (error of the loss function) using $\mathbf{p}^+$;
(2) Calculate the *information matrix* $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ and the *information vector* $\mathbf{b} = \mathbf{J}^T\mathbf{e}$.
(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 [71]: $\mathbf{H}_p = \mathbf{A} - \Lambda\mathbf{M}^{-1}\Lambda^T$, and $\mathbf{r}_p = \mathbf{b}_r - \Lambda\mathbf{M}^{-1}\mathbf{b}_m$.

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

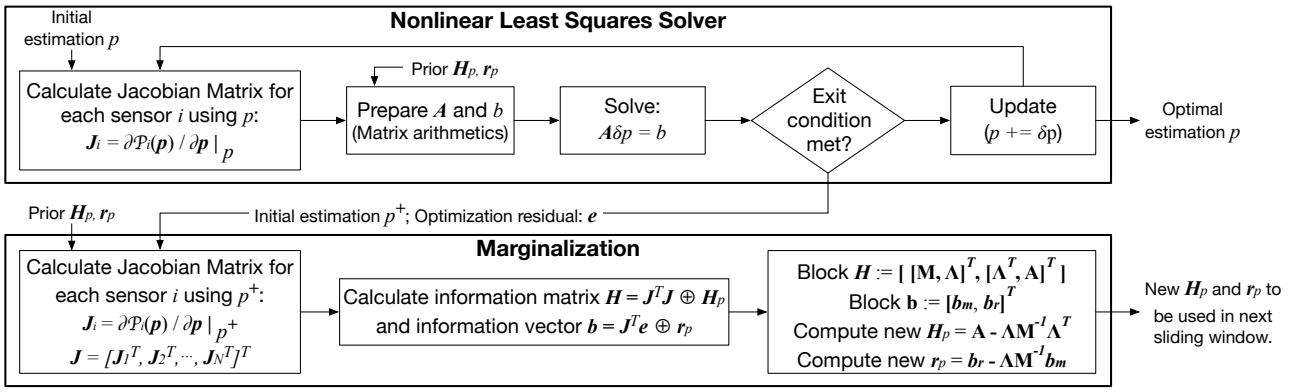


Fig. 2: Algorithm overview of MAP estimation. This high-level algorithm description does not directly translate to a concrete software implementation that can be mapped to hardware. This is because many blocks such as matrix inversion and linear system solver can be implemented in a variety of ways. Archytas generates a concrete implementation for each block using primitive M-DFG nodes and thereby forms a final M-DFG. The final M-DFG reduces the overall computation and facilitates hardware resource sharing by leveraging data sparsity and task dependencies unique to SLAM.

3.2 Generating Macro Data-Flow Graph

3.2.1 Design Overview. Hardware acceleration must target a concrete software implementation, usually represented by a data-flow graph (DFG). In particular, Archytas represents the localization algorithm using a M-DFG, which is a coarse-grained DFG, where each node, instead of being one single operation, is a relatively complex function (e.g., dense matrix multiplication) that executes on a well-optimized hardware block (Sec. 4).

Table 1: Primitive M-DFG nodes.

Node Type	Description
`DMatInv`	Diagonal matrix inversion
`MatMul`	Matrix multiplication
`DMatMul`	Diagonal matrix multiplication
`MatSub`	Matrix subtraction (addition)
`MatTp`	Matrix transpose
`CD`	Cholesky decomposition
`FBSub`	Forward and backward substitution to solve linear system of equations with triangular matrices
`VJac`	Calculate visual Jacobian matrix
`IJac`	Calculate IMU Jacobian matrix

Archytas supports a set of primitive M-DFG nodes listed in Tbl. 1. We choose these primitive nodes because they are low-level enough to build complex algorithms but high-level enough to simplify the M-DFG. M-DFG raises the level of abstraction in representing the software behaviors; it allows us to compose hardware blocks well-optimized for each node/sub-graph and thus greatly simplifies hardware design.

Archytas maps each block in Fig. 2 to one or a set of primitive M-DFG nodes and then forms a final M-DFG. While many blocks are trivially mapped (e.g., calculating **J** is directly mapped to a `VJac` node), other blocks, such as matrix inversion and solving linear systems, can be translated to different combinations of primitive nodes. Archytas generates concrete primitive node combinations for these blocks in a way that reduces the overall computation cost. This is accomplished by leveraging the data sparsity inherent to SLAM to build cost models for potential implementations. We now use specific examples from both the NLS solver and marginalization to explain the cost-driven M-DFG generation.

3.2.2 M-DFG for NLS Solver. The main complexity in the NLS solver is to solve the linear system $\mathbf{A}\delta\mathbf{p} = \mathbf{b}$. A linear system is typically solved using Schur elimination [13, 85], which transforms a system of linear equations of size $(p+q)\times(p+q)$ 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. However, this transformation is not without overhead. Our M-DFG builder determines an optimal p (or equivalently q) and generates a concrete linear system solver to maximize the speedup.

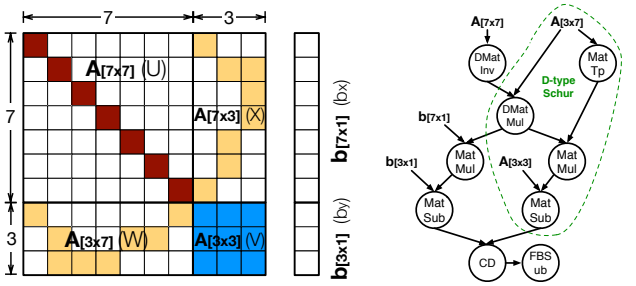
Formally, SLAM requires us to solve the linear system $\mathbf{A}\delta\mathbf{p} = \mathbf{b}$, where $\mathbf{A}$ is a $(p+q)\times(p+q)$ matrix, and both $\delta\mathbf{p}$ and $\mathbf{b}$ are $(p+q)$-dimensional column vectors. Without losing generality, let $\mathbf{A}$ be expressed in the blocked matrix form $\left[\begin{smallmatrix}\mathbf{U} & \mathbf{X}\\ \mathbf{W} & \mathbf{V}\end{smallmatrix}\right]$, where $\mathbf{U}$, $\mathbf{X}$, $\mathbf{W}$, and $\mathbf{V}$ are $p\times p$, $p\times q$, $q\times p$, and $q\times q$ matrices. The original linear system can then be expressed in the following blocked-matrix form:

$$\begin{cases}\mathbf{U}\delta\mathbf{p}_x + \mathbf{X}\delta\mathbf{p}_y = \mathbf{b}_x\\ \mathbf{W}\delta\mathbf{p}_x + \mathbf{V}\delta\mathbf{p}_y = \mathbf{b}_y\end{cases} \tag{3}$$

where $\delta\mathbf{p}_x$ and $\mathbf{b}_x$ are p-dimensional column vectors, and $\delta\mathbf{p}_y$ and $\mathbf{b}_y$ are q-dimensional column vectors.

The idea of Schur elimination is to multiply the first equation with $\mathbf{W}\mathbf{U}^{-1}$ and subtract the first equation from the second equation, which gives a new linear system:

$$\begin{cases}\mathbf{U}\delta\mathbf{p}_x + \mathbf{X}\delta\mathbf{p}_y &= \mathbf{b}_x\\ (\mathbf{V}-\mathbf{W}\mathbf{U}^{-1}\mathbf{X})\delta\mathbf{p}_y &= \mathbf{b}_y - \mathbf{W}\mathbf{U}^{-1}\mathbf{b}_x\end{cases} \tag{4}$$



(a) An example of the blocked A and b. (b) The generated M-DFG.

Fig. 3: The M-DFG builder identifies the best blocking strategy of A and b to generate a concrete M-DFG for solving the system of linear equations.

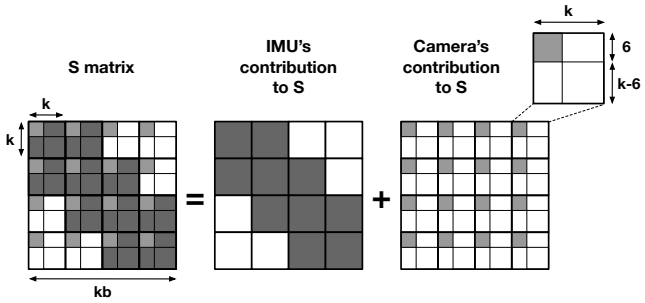


Fig. 4: Storage optimization of the symmetric matrix S.

Critically, the second half of Equ. 4 is a $q \times q$ system and requires solving only $\delta \mathbf{p}_y$, solving which would allow us to solve the first half of Equ. 4, a $p \times p$ system that involves only $\delta \mathbf{p}_x$. Thus, Equ. 4 is much cheaper to solve than Equ. 3. Schur elimination, however, comes with its overhead. Comparing Equ. 3 and Equ. 4, Schur elimination requires computing $\mathbf{W}\mathbf{U}^{-1}\mathbf{X}$ (also known as the Schur complement of matrix $\mathbf{A}$) and $\mathbf{W}\mathbf{U}^{-1}\mathbf{b}_x$. The overhead must be less than the work reduction for this transformation to be meaningful.

The M-DFG builder determines p to minimize the compute cost. This is achieved by building a cost model, parameterized by p, of solving Equ. 4. The cost model is obtained by accumulating the amount of arithmetic operations of each primitive M-DFG node involved in solving Equ. 4 (e.g., matrix multiplication requires n^3 arithmetic operations). The cost minimization yields itself as a simple convex optimization problem that is calculated offline.

Interestingly, the optimal solution almost always blocks $\mathbf{A}$ in such a way that $\mathbf{U}$ is a diagonal matrix. This is because a diagonal matrix $\mathbf{U}$ reduces the computational complexity of inverting $\mathbf{U}$ and calculating $\mathbf{W}\mathbf{U}^{-1}$ from $O(n^3)$ to $O(n)$ and $O(n^2)$, respectively. We call $\mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{X}$ a **D-type Schur** when $\mathbf{U}$ is a diagonal matrix.

As a desirable side effect, the inherent structure of $\mathbf{A}$ dictates that $\mathbf{X}$ is necessarily the same as $\mathbf{W}^T$ when $\mathbf{U}$ is a diagonal matrix [13], which reduces the on-chip memory storage requirement. Fig. 3a shows a blocking example with a 10×10 matrix $\mathbf{A}$. Correspondingly, Fig. 3b shows the generated M-DFG for solving Equ. 4, which includes the sub M-DFG for calculating the D-type Schur.

3.2.3 M-DFG for Marginalization. Most of the blocks in marginalization can be translated to M-DFGs trivially, as they have fixed implementations using the primitive M-DFG nodes. The difficulty lies in transforming the prior calculations ($\mathbf{A} - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{\Lambda}^T$ and $\mathbf{b}_r - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{b}_m$), which have the form of a Schur complement, but can not be simply transformed to a D-type Schur's M-DFG since $\mathbf{M}$ is not a diagonal matrix. Calculating these priors, which we call **M-type Schur**, hinges upon calculating $\mathbf{M}^{-1}$.

Without losing generality, let $\mathbf{M}$ be expressed in the blocked matrix form: $\begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix}$. Then:

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{M}_{11}^{-1} + \mathbf{M}_{11}^{-1}\mathbf{M}_{12}\mathbf{S}'^{-1}\mathbf{M}_{21}\mathbf{M}_{11}^{-1} & -\mathbf{M}_{11}^{-1}\mathbf{M}_{12}\mathbf{S}'^{-1} \\ -\mathbf{S}'^{-1}\mathbf{M}_{21}\mathbf{M}_{11}^{-1} & \mathbf{S}'^{-1} \end{bmatrix}, \quad (5)$$

where $\mathbf{S}'$ is $\mathbf{M}_{22} - \mathbf{M}_{21}\mathbf{M}_{11}^{-1}\mathbf{M}_{12}$.

Archytas, again, builds a cost model to analyze the ideal blocking strategy. We find that the optimal solution blocks $\mathbf{M}$ such that $\mathbf{M}_{11}$ is a diagonal matrix. This way, $\mathbf{S}'$ becomes a D-type Schur and inverting $\mathbf{M}_{11}$, a diagonal matrix, is trivial. The M-DFG is omitted here due to space limit.

A desirable side effect of $\mathbf{S}'$ being a D-type schur is that it can share the same D-type schur hardware resource used by that in the NLS solver ($\mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{W}^T$), as the NLS solver and marginalization are serialized. Archytas schedules both D-type Schur calculations to the same hardware block (Sec. 4.1) to reduce hardware resources.

3.3 Optimizing Data Layout

Along with the M-DFG, Archytas also generates the layout for key data structures to reduce the on-chip storage. We find that the largest gain comes from exploiting data characteristics unique to SLAM. For instance, Fig. 4 shows the structure of the matrix that stores the parameters for the linear system, which we call the S matrix, which contributes to 40% – 80% of the total storage requirement. S is a $kb \times kb$ symmetric matrix, where b is the number of IMU observations in the current sliding window, and k is the number of states in one IMU observation.

Upon an initial glance, it is not clear what opportunity exists other than exploiting the symmetry of the S matrix, which reduces the storage by half. Leveraging the domain knowledge, however, one realizes that S is the sum of two matrices, $\mathbf{S}_c$ and $\mathbf{S}_i$, which represent the contributions of the camera and the IMU, respectively.

The insight is that both $\mathbf{S}_c$ and $\mathbf{S}_i$ are symmetric with structured sparsities. Specifically, the non-zero elements of $\mathbf{S}_i$ exist only in the diagonal and sub/super-diagonal blocks. This is because an IMU observation is related only to the adjacent keyframes. Meanwhile, the non-zero elements of $\mathbf{S}_c$ exist only in a 6×6 sub-block in each $k \times k$ block, where 6 denotes the 6 degrees of freedom [23].

Therefore, Archytas stores $\mathbf{S}_i$ and $\mathbf{S}_c$ separately. Only the diagonal and sub/super-diagonal blocks in $\mathbf{S}_i$ are stored; we compact the non-zero elements in $\mathbf{S}_c$ and exploit the symmetry of the compacted matrix. Overall, our optimization reduces the space requirement from k^2b^2 to $18b^2 + 2bk^2$, which is a 78% space saving under a typical $k = 15$ and $b = 15$. Compared to a CSR-compressed format, our compression consumes 17.8% less space.

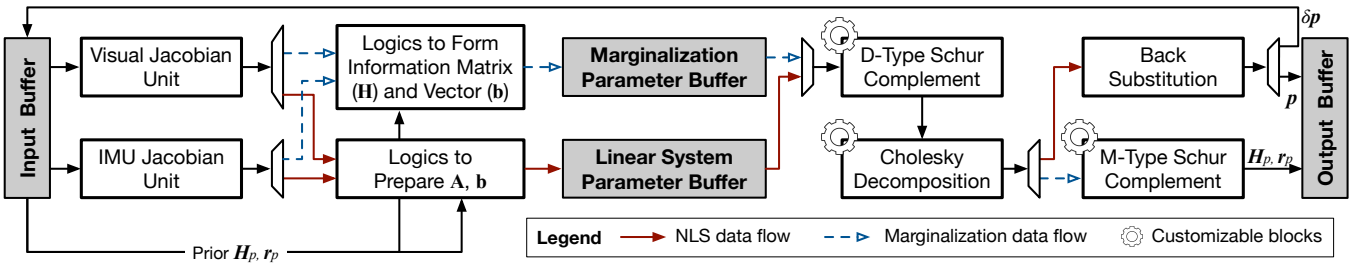


Fig. 5: Overall hardware architecture. Three blocks, D-type Schur, M-type Schur, and Cholesky Decomposition, are parameterized. Solid lines denote the NLS solver data flow, and dash lines denote the marginalization data flow.

4 CUSTOMIZABLE HARDWARE

This section first describes the overall hardware template of Archytas, introducing its customizable design that can be tailored to meet specific resource and/or latency constraints (Sec. 4.1). We then describe how we optimize three key blocks in the template by leveraging the data and computation patterns inherent to SLAM: Jacobian matrix computation (Sec. 4.2), Cholesky decomposition (Sec. 4.3), and Schur complement computation (Sec. 4.4).

4.1 Hardware Design Overview

Fig. 5 shows the hardware architecture, which consists of a collection of optimized hardware blocks, each responsible for a part of the overall M-DFG. For instance, the Visual Jacobian Unit calculates the `VJac` primitive M-DFG node; the D-type Schur Unit calculates the D-type Schur, a sub M-DFG consisting of multiple primitive M-DFG nodes (Fig. 3b).

Customizability To explore the trade-off between performance, energy, and resource utilization, Archytas identifies a set of customizable hardware blocks: their compute and memory resources are parameterized. The exact parameters are determined by the synthesizer depending on the resource/latency constraints (Sec. 5).

Specifically, the customizable blocks are the Cholesky decomposition block, the D-type Schur block, and the M-type Schur complement block. They are ideal candidates for customization, as they present a large resource-vs-latency trade-off space: varying their design parameters could change the end-to-end latency by over 20 × and the overall resource consumption by about 3 × (Sec. 7.2).

Static Scheduling The M-DFG is scheduled to the hardware statically, because the M-DFG is known offline. The scheduler ensures a high hardware utilization through two techniques.

First, the scheduler shares main hardware blocks between the two inherently sequential phases of the algorithm: the NLS phase (solid lines in Fig. 5) and marginalization phase (dashed lines in Fig. 5). The scheduler achieves this by traversing the M-DFG to identify identical subgraphs, which are mapped to the same hardware block. Second, the scheduler pipelines hardware blocks whenever possible. For instance, the Jacobian block and the Schur block are pipelined across feature points.

4.2 Jacobian Matrix Block

The visual matrix calculation is a primitive M-DFG node (Tbl. 1) shared by the NLS solver and marginalization (Fig. 2).

Basic Design Let us explain the basic hardware design using the simple example in Fig. 6. The example is a snapshot of a sample sliding window consisting of 2 keyframes, 3 feature points ($P_1 \sim P_3$), and 4 observations ($O_1 \sim O_4$). Recall that the observations are generated from feature points through the projection function ($\mathcal{P}$ in Equ. 2), whose partial derivatives form the Jacobian matrix, the 3×4 matrix $\mathbf{J}$ in Fig. 6. Only valid <feature point, observation> pairs have non-zero values in the matrix, i.e., need to be calculated.

The basic hardware design is shown in Fig. 7. The Observation block is responsible for calculating the matrix elements (partial derivatives), which requires two pieces of data. To calculate [O_1, P_1], for instance, the Observation block requires: 1) the 3D coordinates of the feature point P_1 in the world coordinate system; this is calculated by the Feature block; and 2) the rotation matrix of keyframe 1 (that contains O_1) with respect to the world coordinate system; this is calculated by the Keyframe block.

Optimizing for Data Reuse We exploit characteristics unique to SLAM data to effectively capture 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). Critically, 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× more feature points than keyframes.

We thus prioritize feature point reuse over rotation matrix reuse. This is naturally achieved by calculating the Jacobian matrix elements in the row-major order. Architecturally, this translates to a "feature-stationary" data flow, where each feature point stays in the Observation block until an entire matrix row is calculated.

Given the "feature-stationary" design, the Feature block and the Observation block form a producer-consumer pair. As a result, their communication is through a FIFO. 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 are stored in a RAM. Note that a RAM is much more power-hungry to access than a FIFO. Had we prioritized rotation matrix reuse (and thus calculating the matrix in a column-major order), the massive amount of feature points would have to be accessed from a power-hungry RAM.

Balancing Pipeline With the Feature block and the Observation block communicating through a FIFO, it is critical to ensure

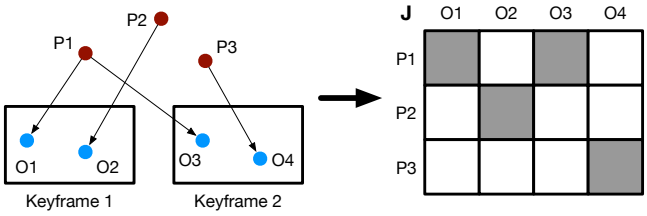


Fig. 6: Illustration of a Jacobian matrix (3×4) for a simple sliding window with 2 keyframes, 3 feature points ($P_1 \sim P_3$), and 4 observations ($O_1 \sim O_4$).

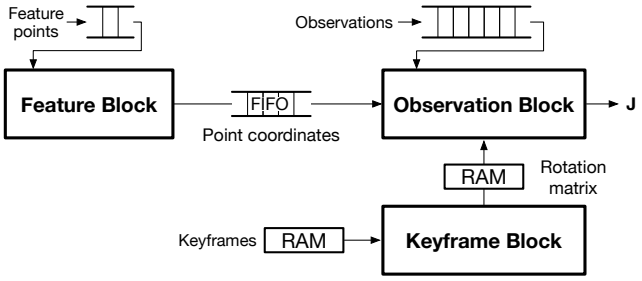


Fig. 7: Overall architecture of Jacobian matrix calculation, which is designed to exploit feature point reuse.

that speeds of the two blocks roughly match so as to avoid a large FIFO or constantly stalling the pipeline.

We observe that the number of observations is typically 10× more than that of feature points. Thus, the Observation block does more computation, and needs to be more aggressively pipelined, than the Feature block to have an equal speed. However, it is impossible for a design to always ensure a balanced pipeline all the time. This is because the amount of work the Observation block does for each feature point is non-deterministic, as each feature point is naturally associated with a different amount of observations.

Our design decision is to build a *statistically-balanced* pipeline design by statically pipelining the blocks based on the *average* statistics profiled offline. This empirical decision greatly simplifies 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. 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.

Analytical Modeling Given the statistically-balanced pipeline design above, the total latency of calculating the Jacobian matrix elements for a feature point L_{Jac}, ignoring the pipeline start-up delay, is given by:

$$L_{Jac} = N_oC_o. \tag{6}$$

4.3 Cholesky Decomposition Block

Cholesky decomposition is a primitive M-DFG node (Tbl. 1) that participates in solving linear systems and inverting matrices. While Cholesky decomposition is not unique to SLAM, Archytas leverages SLAM-specific knowledge to optimize its hardware design.

Recall that Cholesky decomposition decomposes a symmetric matrix S into LL^T, where L is a lower triangular matrix. The hardware iteratively generates columns of L, starting from the first column. Assuming that S is of dimension $n \times n$, the first iteration would calculate the first column of L (the Evaluate phase), using which a new matrix S' of size $(n-1) \times (n-1)$ is generated (the Update phase). S' becomes the input matrix to the second iteration. This process continues until the entire L is calculated.

Fig. 8 shows the basic hardware design of the Cholesky decomposition unit, which cascades the Evaluate structure and the Update structure. The example shows how a 6×6 input matrix S (denoted as S_6), after the two phases, becomes a 5×5 matrix S_5, which becomes the input of the next iteration.

Optimizing and Parameterizing the Design Analyzing the fine-grained data dependencies shows that the Evaluate phase and the Update phase could be pipelined [48]. However, that Update phase is longer than the Evaluate phase. Specifically, at iteration i the number of operations of the Evaluate and the Update phase is i and $i(i-1)/2$, respectively.

To ensure a balanced pipeline, our design uses multiple Update units, the exact number of which is parameterized. Fig. 9 shows a sample hardware design with six Update units, which are time-multiplexed with the Evaluate unit. Fig. 10 shows the timeline of execution under this particular configuration, where E_i and U_i denote the Evaluate and Update latency in iteration i, respectively.

Analytical Modeling Let us use the example in Fig. 10 to explain the analytical latency model. Given 6 Updates units, every 6 Evaluate-Update iterations form a round; next round could only start when there is no structural hazard. That is, the Evaluate unit and at least one Update unit are both available, which leads to two scenarios: 1) when an Update unit is available later than the Evaluate unit is available (e.g., Round 1), and 2) when the Evaluate unit is available later than an Update unit (e.g., Round 2).

Generally, given $\mathbf{s}$ Update units, the $m \times m$ input matrix S, and the latency of the Evaluate unit E, the total latency is:

$$L_{cholesky} = \sum_{k=0}^{\lfloor \frac{m}{s} \rfloor} max\{\mathbf{s}E, E + \frac{m_k(m_k-1)}{2}\}, \tag{7}$$

$$m_k = m - sk - 1 \tag{8}$$

where the *max* operation models the two scenarios.

4.4 Schur Complement Blocks

Calculating the Schur complement is central to both the NLS solver and marginalization: the NLS solver calculates a D-type Schur $\mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{W}^T$, and the marginalization calculates a M-type Schur $\mathbf{A} - \mathbf{\Lambda}\mathbf{M}^{-1}\mathbf{\Lambda}^T$. The difference is that $\mathbf{U}$ is a diagonal matrix that can be trivially inverted, while $\mathbf{M}$ is a generic matrix, which is inverted according to Equ. 5.

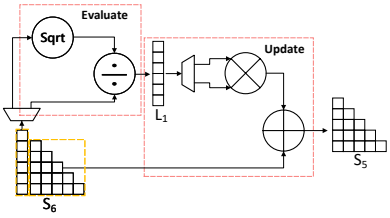


Fig. 8: The microarchitecture of a simple Cholesky decomposition hardware structure with one Evaluate and one Update unit.

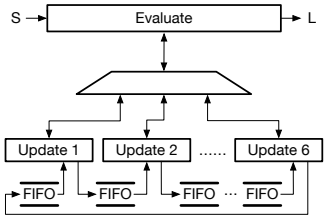


Fig. 9: To ensure a balanced pipeline between Evaluate and Update, our Cholesky decomposition unit uses one Evaluate unit and multiple (six here) time-multiplexed Update units.

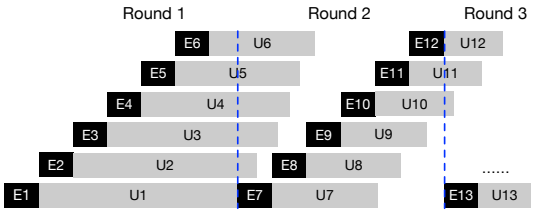


Fig. 10: The execution timeline under the hardware configuration in Fig. 9. E_i and U_i denote the Evaluate and Update latency in iteration i, respectively. We show three rounds of execution; each round finishes 6 Evaluate and Update phases, where 6 is the number of Update units.

The basic hardware design for both types is a straightforward mapping from their M-DFGs. For instance, the D-type Schur hardware cascades blocks for `DMatInv` ($\mathbf{U}^{-1}$), `DMatMul` ($\mathbf{W}\mathbf{U}^{-1}$), `MatMul` ($\mathbf{W}\mathbf{U}^{-1}\mathbf{W}^T$), and `MatSub` ($\mathbf{V} - \mathbf{W}\mathbf{U}^{-1}\mathbf{W}^T$). We observe that the D-type Schur of different feature points are independent. We exploit the parallelism by pipelining the D-type Schur unit across the feature points. The M-type Schur unit meanwhile is inherently not amenable to this feature-level pipelining, as it operates on a matrix ($\mathbf{M}$) that mixes information from all the feature points.

Analytical Modeling The two hardware Schur blocks are parameterized by the number of MAC units, which dictates the performance of `MatMul` — the bottleneck of calculating both Schur complements.

Assuming that each feature point on average has N_o observations, the D-type Schur module multiplies a $6N_o \times 1$ vector ($\mathbf{W}\mathbf{U}^{-1}$) with $1 \times 6N_o$ vector ($\mathbf{W}^T$). Given the number of MAC units $\mathbf{n_d}$, the latency of the calculating the D-type Schur complement for a feature point is:

$$L_{DSchur}(\mathbf{n_d}) = (6N_o)^2/\mathbf{n_d} \tag{9}$$

Given that there are $\mathbf{n_m}$ MAC units in the M-type hardware, the overall M-type Schur latency is the following:

$$\begin{aligned} L_{MSchur}(\mathbf{n_m}) \approx & 15a_m + a_m^2 + b_k(15 + a_m)(6(b-1)+9) \\ & + b_k(6(b-1)+9)^2,\ b_k = \frac{15 + a_m}{\mathbf{n_m}} \end{aligned} \tag{10}$$

where a_m denotes the number of features that are moved out of the current sliding window (to be marginalized), and b denotes the number of keyframes in the sliding window. The approximation is introduced by eliding trivial operations. We omit the detailed derivation due to space limit.

5 SYNTHESIZING THE HARDWARE

Instead of designing just one specific accelerator, Archytas allows designers to explore a large design space. In particular, designers specify the resource, latency, or power constraints, from which Archytas generates an optimal accelerator design that meets the constraints. This is accomplished by calculating the parameters of the three customizable blocks: the number of Update units $\mathbf{s}$ in the Cholesky decomposition block and the number of MAC units in the D-type Schur and the M-type Schur blocks, $\mathbf{n_d}$ and $\mathbf{n_m}$, respectively.

Problem Formulation The task of hardware generation is expressed in the form of a constrained optimization:

$$\begin{aligned} & \min_{\mathbf{n_d},\mathbf{n_m},\mathbf{s}} \ Power(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) \\ & s.t.\ Lat(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) \le L^*,\ Res(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) \le R^*, \end{aligned} \tag{11}$$

where $Power(\cdot)$, $Lat(\cdot)$, and $Res(\cdot)$ denote the total power, latency, and resource utilization, respectively; they are functions of $\mathbf{n_d}$, $\mathbf{n_m}$, and $\mathbf{s}$. L^* is the latency constraint specified by the designer, and R^* is the resource constraint imposed by a particular FPGA system.

Note that other optimization formulations are possible. For instance, the following formulation could be used for scenarios where performance, rather than power, is the main design objective:

$$\min_{\mathbf{n_d},\mathbf{n_m},\mathbf{s}} \ Lat(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s})\ s.t.\ Res(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) \le R^*, \tag{12}$$

Latency Model Archytas derives the latency model by calculating the critical path latency of the M-DFG given the analytical latency models of each of the primitive nodes:

$$Lat(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) = Iter \times L_{NLS}(\mathbf{n_d}, \mathbf{s}) + L_{Marg}(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) \tag{13}$$

where L_{NLS} denotes the latency of an iteration of the (iterative) NLS solver, $Iter$ denotes the total number of iterations in the NLS solver — a parameter set by the application, and L_{Marg} denotes the marginalization latency.

The critical-path latency of an NLS iteration (the blocks along the solid arrows in Fig. 5) is expressed as follows:

$$L_{NLS}(\mathbf{n_d}, \mathbf{s}) = \sum_{i=1}^{a} \max\{L_{Jac}, L_{DSchur}(\mathbf{n_d})\} + L_{Cholesky}(\mathbf{s}) + L_{Sub} \tag{14}$$

where a denotes the number of feature points in the current sliding window, L_{Jac} denotes the latency to calculate the Visual Jacobian matrix (Equ. 6), $L_{DSchur}(n)$ denotes the latency to calculate the D-type Schur complement and is parameterized by $\mathbf{n_d}$ (Equ. 9), $L_{Cholesky}(s)$ is the latency of Cholesky decomposition and is parameterized by $\mathbf{s}$ (Equ. 7), and L_{sub} denotes the back substitution latency, which is fixed-function logic with a fixed latency independent of $\mathbf{n_d}$ and $\mathbf{s}$. The *max* operation reflects the pipeline parallelism between calculating the Visual Jacobian and the D-type Schur complement across all a feature points.

The marginalization latency is the cumulative latency of calculating the Jacobian matrix, D-type, and M-type Schurs:

$$L_{Marg}(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) = a_m L_{Jac} + L_{DSchur}(\mathbf{n_d}) + L_{Cholesky}(\mathbf{s}) + L_{MSchur}(\mathbf{n_m}) \quad (15)$$

where a_m denotes the number of feature moved out of the current sliding window (to be marginalized).

Resource Model The resource consumption of the hardware is modeled as:

$$Res(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) = R_0 + \mathbf{n_d} \times R_d + \mathbf{n_m} \times R_m + \mathbf{s} \times R_s, \quad (16)$$

where R_d, R_m, and R_s denote the *unit* resources consumption of the structured that are parameters by $\mathbf{n_d}$, $\mathbf{n_m}$, and $\mathbf{s}$, respectively, and R_0 denotes the resource consumption independent of customization. We consider four major types of resources when targeting FPGA: LUT, FF, BRAM, and DSP. The resource constraint must be met for all four resource types, because exceeding the limit of even one resource type would mean that the design could not be instantiated.

Power Model The power of an FPGA is dictated by its hardware resource utilization, similar to the fact that an ASIC's power is correlated with its area. Thus, we model the power as the weighted sum of the three customization parameters ($\mathbf{n_d}$, $\mathbf{n_m}$, and $\mathbf{s}$) plus a constant base power P_0 that is independent of customization:

$$Power(\mathbf{n_d}, \mathbf{n_m}, \mathbf{s}) = P_0 + \mathbf{n_d} \times P_d + \mathbf{n_m} \times P_m + \mathbf{s} \times P_s, \quad (17)$$

We fit the coefficients (P_d, P_m, and P_s) for a particular FPGA platform 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.

Synthesizer The structure of the optimization problem in Equ. 11 is a 3-variable mixed-integer convex programming, for which a near-optimal solution can be found in milliseconds using common optimization packages such as YALMIP [47] and MLOPT [16].

The solver calculates $\mathbf{s}$, $\mathbf{n_d}$, and $\mathbf{n_m}$, which are then used to generate the three customizable blocks. The synthesizer will also automatically customize the on-chip memory sizes for data internal to the three customized blocks given the $\mathbf{s}$, $\mathbf{n_d}$, and $\mathbf{n_m}$ values, and add memory interfaces and the address generation modules.

6 DYNAMIC OPTIMIZATIONS

Given an initial design generated by the hardware synthesizer, Archytas dynamically optimizes the hardware configuration *at run time* to save power. We first motivate the need for run-time optimizations (Sec. 6.1), and then describe our run-time system design and implementation (Sec. 6.2).

6.1 Opportunity

The discussion so far assumes that the amount of work is fixed over time. However, in a real-world SLAM deployment the amount of work done in each sliding window often varies over time in order to sustain the accuracy. If a sliding window has fewer feature points, the accuracy would drop because less information is available to estimate the pose. Fig. 11 shows a snapshot of execution on the KITTI Odometry dataset [26]. The error (left y-axis) increases as the number of feature points (right y-axis) decreases.

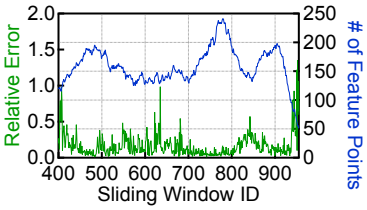


Fig. 11: Fewer feature points (right y) generally leads to higher relative error (left y).

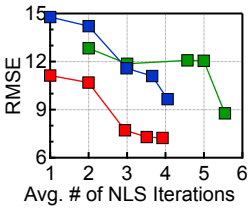


Fig. 12: More NLS iterations (x) lowers the error rate (y).

A common strategy to sustain the accuracy level, when the feature points are insufficient, is to increase the number of iterations in the iterative NLS solver (i.e., *Iter* in Equ. 13). Using profiling results from the KITTI dataset [26], Fig. 12 shows that the overall Root Mean Square Error (y-axis) decreases as the number of NLS solver iterations (x-axis) increases.

More iterations lead to more work and, thus, require more capable hardware configurations. Our static design is necessarily conservative to accommodate a large iteration count. At run time, if fewer NLS iterations are needed Archytas re-configures the hardware to provision less hardware resources.

6.2 Design

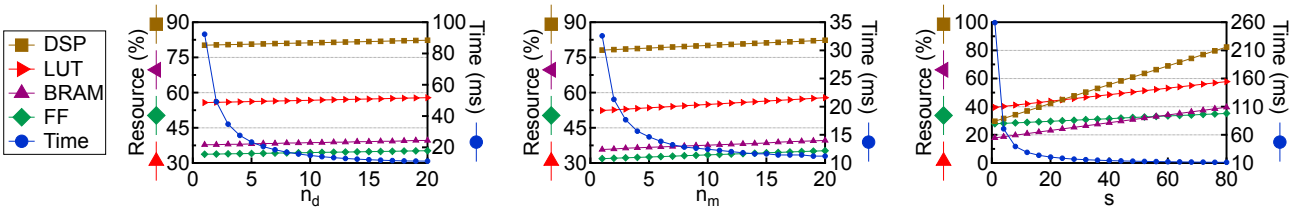
The idea is that Archytas dynamically adjusts the number of NLS iterations *Iter* required to sustain a target accuracy; given an *Iter*, Archytas calculates a new hardware configuration. We will show that with intelligent design decisions, the run-time reconfiguration has little to none overhead.

Adjusting Iteration Count We use a simple mechanism to dynamically adjust the NLS iteration count. We first construct a lookup table that maps the number of feature points to *Iter*. This is done by profiling datasets of interest offline. *Iter* is capped at 6, because we find that iterating over 6 times usually provides little accuracy improvement.

In a real-world deployment, when our system enters a new environment we would collect and profile the data from the environment to determine the *Iter* cap. The profiling and optimization can happen asynchronously and does not have to be interactive. The *Iter* cap can then be used later when the system enters the same environment. This strategy is commonly used in robotics, where a robot collects and analyzes data from a new environment and creates an optimized strategy for use in the future [50].

At run time, the sensing front-end provides the number of feature points, which is then mapped to *Iter* based on a simple two-bit saturating counter. That is, *Iter* is adjusted (incremented or decremented) when the number of feature points maps to a different *Iter* (according to the offline-constructed lookup table) in two consecutive sliding windows.

Re-optimization Given an *Iter*, the run-time system obtains a new latency model (which is a function of *Iter* as shown in Equ. 13) and solves the following optimization problem to generate a new



(a) Impact of n_d. (b) Impact of n_m. (c) Impact of s.

Fig. 13: The influences of n_d, n_m, s on the hardware resources (left y-axes) and execution time (right y-axes).

hardware configuration:

$$\min_{\mathbf{n_d},\mathbf{n_m},\mathbf{s}} \; Power(\mathbf{n_d},\mathbf{n_m},\mathbf{s})$$
$$s.t.\; Lat(\mathbf{n_d},\mathbf{n_m},\mathbf{s}) \leq L^*, \mathbf{n_d} < \mathbf{n_d}^*, \mathbf{n_m} < \mathbf{n_m}^*, \mathbf{s} < \mathbf{s}^*, \qquad (18)$$

where $\mathbf{n_d}^*$, $\mathbf{n_m}^*$, and $\mathbf{s}^*$ denote the initial resource allocations generated by the static synthesizer.

Examining the optimization formulation, we see that the run-time optimization enforces that the new hardware must consume *less resource than the initial design on all resource types* (i.e., the customization parameters must take smaller values). This is critical as it allows us to use simple mechanisms such as clock-gating to reconfigure/throttle the hardware. Otherwise, a complete reconfiguration of the FPGA (synthesis, layout, bitstream programming) would have to take place, impractical for real-time operations.

Due to this design decision, we can avoid reconfiguring the FPGA dynamically, i.e., no new bitstream is sent to the FPGA at run time. Each sliding window, the host triggers the FPGA and simply passes three numbers ($\mathbf{n_d}$, $\mathbf{n_m}$, $\mathbf{s}$), which toggle the clock gating logic according to a look-up table. Critically, since there are only 6 $Iter$ choices, Equ. 18 is solved exhaustively against all $Iter$ values *offline*. At run time before the FPGA is triggered for each sliding window, the $\mathbf{n_d}$, $\mathbf{n_m}$, and $\mathbf{s}$ values are looked up based on $Iter$ and passed to the FPGA if different from the current values. This way, run-time re-optimization effectively has no overhead.

Discussion Our contribution here is two-fold. First, we are the first to identify the NLS iteration count as a run-time knob. Prior localization accelerators do not harness this knob as they treat the NLS optimization as a black box. By identifying this knob, we *expose the opportunity for clock gating*. Without identifying the knob, the hardware would not clock gate the circuits during iterations that could have been omitted.

Second, we propose a simple mechanism (2-bit saturating counter) to tune the knob and show that this simple mechanism could readily reduce power with little accuracy impact. We leave it to future work to explore other mechanisms to tune the knob (e.g., training a machine learning model).

7 EVALUATION

After the experimental setup (Sec. 7.1), we show that the three customizable design parameters indeed provide a large latency-vs-resource trade-off space (Sec. 7.2), which can be explored by our hardware generator extremely rapidly (Sec. 7.3). The generated accelerators achieve one order of magnitude higher speed and/or lower energy compared to the baselines (Sec. 7.4) and other accelerators (Sec. 7.5). We show that our dynamic optimization reduces energy by double digits without affecting the accuracy (Sec. 7.6). Finally, we show results on other FPGA platform and algorithms to demonstrate the general applicability of Archytas (Sec. 7.7).

7.1 Experimental Setup

Hardware Platform We target the Xilinx Zynq-7000 SoC ZC706 FPGA [7]. The synthesizer is implemented in Python, and the generated Verilog code goes through the FPGA synthesis and layout flow using Vivado Design Suite 2018.2.

The FPGA is triggered by the host for each sliding window. The host passes to the FPGA the visual features from the sensing front-end as well as the three customization parameters if they are different from the previous sliding window.

Our FPGA designs operate at a fixed frequency of 143 MHz. The FPGA power consumption is estimated using the Vivado power analysis tool using real workloads under test. All power and resource consumption data is obtained after the designs pass the post-layout timing.

Implementation Details Our synthesizer is implemented in Python, which generates synthesizable Verilog code. The synthesizer uses the popular YALMIP package [47] to solve the constrained optimization problem.

Dataset We evaluate Archytas using two common datasets: KITTI Odometry [26] (grayscale sequence) for self-driving cars and EuRoC [14] (Machine Hall sequences) for drones.

Baselines We compare against a software implementation of SLAM, which uses Google's ceres solver [2] to implement bundle adjustment. The software implementation 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 [4] 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. We also compare against previous localization accelerators.

7.2 Impact of Customization

The three knobs, $\mathbf{n_d}$, $\mathbf{n_m}$, $\mathbf{s}$, significantly influence the latency and the overall resource consumption. They are, thus, ideal targets for customization. Fig. 13 shows the influences of $\mathbf{n_d}$, $\mathbf{n_m}$, $\mathbf{s}$ on the FPGA

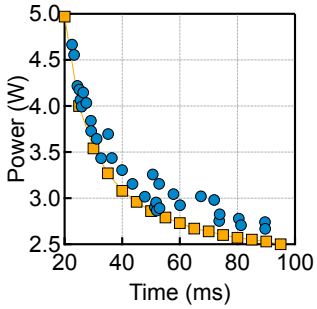


Fig. 14: Latency-vs-power Pareto optimal frontier of power-optimized designs generated by Archytas.

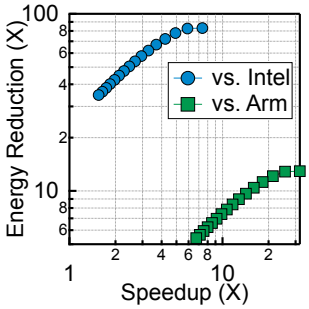


Fig. 15: Speedup and energy reduction on a KITTI trace using Pareto-optimal designs from Fig. 14.

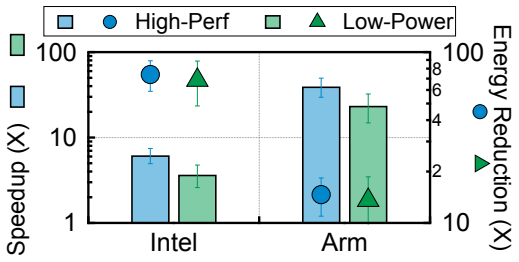


Fig. 16: Speedup and energy reduction of the two Archytas variants over the baselines. Error bars denote one stdev. No design in this figure uses the dynamic optimization.

resources (left y-axes) and execution time (right y-axes). Each figure sweeps one parameter while leaving the other two constant.

Overall, increasing the three customization knobs ($\mathbf{n_d}$, $\mathbf{n_m}$, $\mathbf{s}$) leads to significantly better performance until the point of diminishing return. $\mathbf{s}$ has the most significant impact on resource utilization (e.g., 50% increase in the DSP utilization when $\mathbf{s}$ increases from 1 to 80). The DSP is the most demanded resource, because many hardware blocks require intensive matrix calculations.

7.3 Hardware Generator Efficiency

The three parameters ($\mathbf{n_d}$, $\mathbf{n_m}$, $\mathbf{s}$) constitute a design space of about 90,000 designs. If one were to exhaustively search the design space, going through the FPGA synthesis and layout flow (which takes roughly 1.5 hours on our machine), to identify the optimal design point, it would take 15 years.

Our hardware generator takes around 3 seconds to identify a design given the resource constraint and generate the synthesizable Verilog code, which then goes through the FPGA synthesis and layout flow, as any FPGA design would do.

Validation It would be nearly impossible to definitively show that the designs generated by our synthesizer are Pareto-optimal, as it would require an exhaustive search of the design space, which is prohibitively expensive.

We perform a best-effort validation. Fig. 14 shows the latency-vs-power Pareto-optimal designs (square markers) *generated by our hardware generator* by varying the latency constraint in Equ. 11. We then slightly vary the parameters of the designs on the frontier, and measure their corresponding latencies and power consumptions; the results are denoted by the circle markers. The circle markers are Pareto-dominated by the square markers, suggesting the validity of the Pareto-optimal frontier and, by extension, our generator.

7.4 Speedup and Energy Reduction Results

Representative Study Using a KITTI trace, Fig. 15 showcases the speedup and energy reduction of the power-optimized, Pareto-optimal designs in Fig. 14 over both baselines. A higher speedup leads to a higher energy reduction due to the impact of static power, but the energy reduction increase eventually tapers off, because a design that consumes overly high power provides diminishing return for speedup.

The speedup over Comet Lake is lower than that over Arm, but the energy reduction is higher. The best design achieves 7.4× speedup and 83.1× energy reduction over Comet Lake and 32.0× speedup and 12.9× energy reduction over Arm.

Comprehensive Study The significant efficiency gains are consistently found in the entire KITTI and EuRoC benchmarks. We select two design points that are optimized under a 20 ms and a 33 ms latency constraint, respectively. We denote them High-Perf and Low-Power. The former consumes about 2 W higher power than the latter. The difference between the two designs are evident in Tbl. 2, which compares the FPGA resource consumption and the three customization parameter values of the two designs. High-Perf consumes more resources than Low-Power. High-Perf is ultimately limited by the DSP resource.

Fig. 16 summarizes the average speedup and energy reduction of these two designs over the baselines on the EuRoC and KITTI benchmarks. Error bars denote one standard deviation across benchmark traces. High-Perf achieves a 6.2× speedup and 74.0× energy reduction over Intel and a 39.7× speedup and 14.6× energy reduction over Arm. Low-Power achieves a 3.7× speedup and 68.6× energy reduction over Intel and a 23.6× speedup and 13.6× energy reduction over Arm.

Table 2: FPGA resource consumption (utilization percentages and absolute numbers) and the customization parameter values of the High-Perf and Low-Power designs.

Design	LUT	FF	BRAM	DSP	$\mathbf{n_d}$	$\mathbf{n_m}$	$\mathbf{s}$
High-Perf	62.41% (136432)	37.28% (163006)	46.88% (255.5)	94.33% (849)	28	19	97
Low-Power	43.81% (95777)	28.97% (126670)	26.79% (146)	49.11% (442)	21	8	34

7.5 Existing Accelerator Comparisons

Different localization accelerators target slightly different algorithmic variants/FPGA platforms and are evaluated on different benchmarks, so a fair comparison is difficult. Below is our best-effort comparisons with prior localization accelerators.

Apart from the efficiency gains we will show next, Archytas has two fundamental advantages. First, Archytas is a *synthesis* framework that generates a specific accelerator instance given the latency/power/resource specifications, while prior work manually designs a specific accelerator instance with a fixed resource-vs-latency-vs-power trade-off. Second, our generated accelerator also adapts to run-time dynamisms while all prior accelerators do not.

π-BA [45] accelerates only the Jacobian calculation and Schur elimination on FPGA; BAX [75] is a full hardware accelerator for BA; neither supports marginalization and both are evaluated on the BAL dataset [8]. To normalize the difference between datasets, we report the results per NLS solver iteration. High-Perf achieves a 137× speedup and 132× energy reduction over π-BA and is 9 × faster and consumes 44% less energy compared to BAX. The gains over BAX mainly come from our optimized datapath in the hardware, whereas BAX uses generic vector units for acceleration.

Zhang et al. [88] jointly explore the algorithm and hardware design space for accelerating localization. They instantiate an FPGA accelerator, where the optimization problem is solved using an on-manifold Gauss-Newton (GN) method (as opposed to the LM algorithm by Archytas). High-Perf uses roughly 2× more hardware resources [87] and achieves over 20× speedup on EuROC. The main gain of Archytas, from our educated guess, comes from that Archytas generates an optimal M-DFG configuration for the NLS solver to reduce the total computation while Zhang et al. use a fixed parameter configuration for the NLS solver.

HLS Comparison Two graduate students with deep experience in HLS development spent about a week on optimizing a Vivado HLS implementation of Cholesky decomposition (Sec. 4.3). The HLS-generated design operates at 30% lower clock frequency, roughly doubles the resource consumption, and overall is 16.4× slower than our optimized design. The performance gains over HLS comes from how we co-design the Cholesky decomposition algorithm to expose fine-grained optimization opportunities that HLS does not exploit (Fig. 10). Specifically, we expose 1) the pipeline parallelism between the Evaluate and Update stage, and 2) the independency among different Update iterations.

PISCES [9] is an HLS-based FPGA accelerator for the entire SLAM pipeline, including feature extraction and BA. Comparing the BA part alone, High-Perf is about 5.4× faster on the MH sequences in EuRoC with about 3× higher energy, based on data estimated from the figures in PISCES [9].

Fundamentally, HLS requires extensive manual tuning of *each module* in the algorithm with knowledge in both algorithms and hardware. The manual tuning effort exponentially increases when designers must meet prescribed design specifications (performance, power, resource targets). In contrast, Archytas requires only a high-level algorithm description from users and automatically generates *the entire accelerator* given the design specification.

7.6 Dynamic Optimizations

Our run-time system effectively reduces the energy. Using High-Perf, dynamically clock-gating the unused hardware structures leads to 21.6% energy saving on the KITTI dataset and 20.8% on the EuRoC dataset. Using Low-Power, the energy saving is 7.7% on KITTI and 6.8% on EuRoC. Note that the dynamic optimization involves only table lookups (Sec. 6.2) and, thus, has little to none overhead.

The dynamic optimization has minimal impact on accuracy. Across all KITTI traces, we see no increase in the mean translational accuracy. Across all EuRoC traces, the mean translational accuracy is degraded by at most 0.01 cm. In many traces, dynamic optimization even improves the accuracy due to the stochastic nature of the iterative optimization.

Finally, the results in Fig. 16 are obtained when both the baselines and our accelerators do not use the dynamic optimization. When both use the dynamic optimization, High-Perf achieves a 5.1× speedup and 89.8× energy reduction over Intel and a 30.4× speedup and 41.3× energy reduction over Arm; Low-Power achieves a 2.8× speedup and 62.2× energy reduction over Intel and a 16.7× speedup and 28.5× energy reduction over Arm.

7.7 Results on Other FPGAs and Algorithms

To show the generalizability of Archytas, we evaluate two additional Xilinx FPGAs: one from the Kintex-7 series (XC7K160tfbg484) and the other from the Vertix-7 series (XC7VX690tffg1761). We use Archytas to generate the biggest design that fits each board. On the EuROC dataset, our designs achieve a 6.6× and 10.2× speedup as well as a 105.1× and 114.6× energy reduction over Intel on the two boards, respectively. The speedups over Arm are 56.2× and 86.3×, and the energy reduction over Arm are 68.9× and 75.1×, on the two boards, respectively.

While this paper focuses on MAP in SLAM, MAP is prevalent in other robotic algorithms [6, 17] such as planning [18], control [82], and tracking [64]. To demonstrate how Archytas generalizes to non-SLAM tasks, we target two new algorithms. For both algorithms, we use their well-optimized parallel software implementations [2] as the baselines and use Archytas to generate the fastest accelerator that fits the ZC706 FPGA for each algorithm.

First, we target the curve fitting problem used for robotic planning [18, 30]. Our design achieves an 8.5× speedup and a 257.0× energy reduction compared to Intel. Second, we target the pose estimation problem commonly found in Augmented Reality [52]. Our design achieves an 7.0× speedup and a 124.8× energy reduction compared to Intel.

8 RELATED WORK

Localization Accelerators Prior accelerators target both ASIC [42, 73, 83] and FPGA [9, 11, 24, 25, 45, 46, 86, 88], and accelerate different algorithms including optimization-based algorithms [21, 22, 58, 65] (e.g., BA) and probabilistic estimation-based filtering algorithms [35, 51, 53, 57]. This paper focuses on BA-based algorithms due to their wide applicability, and target FPGA due to its rich sensor interfaces that ease its integration into an end-to-end computing system [84].

Archytas differs from previous accelerators in two main ways. First, unlike most of the prior accelerators, which are manually designed for a particular design point, Archytas is a localization accelerator generation framework, which generates an accelerator given the design specifications. Second, Archytas-generated accelerators dynamically re-optimize themselves at run time to adapt to the operating environment, where all prior accelerators are static.

Archytas is part of a broad, recent effort in designing an efficient computing substrate for robotic computing [78]. Recent efforts also include accelerating motion planning and control [43, 59–61, 68], exploring the general design space [31], leveraging distributed platforms [32, 33, 55], resource management [10, 89], and benchmarking [12, 29, 79].

Accelerator Design Designing, generating, and scheduling accelerators is a thriving area of research. Recent efforts include new programming models/intermediate representations [20, 38, 62, 69] and design space exploration/scheduling through search [80, 81] and constrained optimization [34, 63].

Different from prior work, Archytas abstracts the software behavior using a coarse-grained M-DFG, which raises the level of abstraction and simplifies the hardware design. The coarse-grained abstraction is critical because, unlike prior work that targets relatively lean computational kernels (e.g., FFT, convolution), we target robotics algorithms (i.e., localization) that are very complex with hundreds of thousands of lines of code. M-DFG allows us to compose accelerators well-optimized for individual kernels.

While M-DFG has been demonstrated as a desirable abstraction [49, 68], we show that deriving a M-DFG for localization is non-trivial because key algorithmic kernels have a range of implementation choices. We exploit the data sparsity and dependencies unique to SLAM to automatically generate a concrete M-DFG from the high-level algorithm description.

9 CONCLUSION

Using MAP as a case study, this paper provides a concrete example of automatically generating accelerators for robotic algorithms, which are becoming increasingly complex. The key is to use a coarse-grained M-DFG to represent the MAP algorithm, which simplifies the hardware design and translates hardware generation into a constrained optimization that can be solved in seconds. We also demonstrate a lightweight run-time system that exploits environment dynamism to reduce energy by re-optimizing the accelerator.

ACKNOWLEDGMENTS

We thank the anonymous reviewers from MICRO 2021 for their valuable feedback and/or guidance. Qiang Liu and Jie Tang are the corresponding authors of the paper.

REFERENCES

[1] [n.d.]. ARCore fundamental concepts. https://developers.google.com/ar/discover/concepts
[2] [n.d.]. Ceres Solver. http://ceres-solver.org/
[3] [n.d.]. Ceres Users. http://ceres-solver.org/users.html
[4] [n.d.]. TX1 datasheet. http://images.nvidia.com/content/tegra/embedded-systems/pdf/JTX1-Module-Product-sheet.pdf
[5] [n.d.]. VINS-Fusion. https://github.com/HKUST-Aerial-Robotics/VINS-Fusion
[6] [n.d.]. What are Factor Graphs? https://gtsam.org/2020/06/01/factor-graphs.html
[7] [n.d.]. Xilinx Zynq-7000 SoC ZC706 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html
[8] Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. 2010. Bundle adjustment in the large. In *European conference on computer vision*. Springer, 29–42.
[9] Bahar Asgari, Ramyad Hadidi, Nima Shoghi, and Hyesoon Kim. 2020. PISCES: Power-aware implementation of SLAM by customizing efficient sparse algebra. In *Design Automation Conference*. IEEE, 1–6.
[10] Sabur Baidya, Yu-Jen Ku, Hengyu Zhao, Jishen Zhao, and Sujit Dey. 2020. Vehicular and edge computing for emerging connected and autonomous vehicle applications. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[11] Konstantinos Boikos and Christos-Savvas Bouganis. 2016. Semi-dense SLAM on an FPGA SoC. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.
[12] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. 2018. MAVBench: Micro aerial vehicle benchmarking. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 894–907.
[13] Duane C Brown. 1958. *A solution to the general problem of multiple station analytical stereotriangulation*. D. Brown Associates, Incorporated.
[14] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. 2016. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research* 35, 10 (2016), 1157–1163.
[15] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. 2016. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics* 32, 6 (2016), 1309–1332.
[16] Abhishek Cauligi, Preston Culbertson, Bartolomeo Stellato, Dimitris Bertsimas, Mac Schwager, and Marco Pavone. 2020. Learning mixed-integer convex optimization strategies for robot planning and control. In *Conference on Decision and Control (CDC)*. IEEE, 1698–1705.
[17] Frank Dellaert, Michael Kaess, et al. 2017. Factor graphs for robot perception. *Foundations and Trends® in Robotics* 6, 1-2 (2017), 1–139.
[18] Jeremie Deray, Bence Magyar, Joan Solà Ortega, and Juan Andrade-Cetto. 2019. Timed-elastic smooth curve optimization for mobile-base motion planning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Institute of Electrical and Electronics Engineers (IEEE), 3143–3149.
[19] K Di, J Wang, S He, B Wu, W Chen, R Li, LH Matthies, and AB Howard. 2008. Towards autonomous Mars rover localization: Operations in 2003 MER mission and new developments for future missions. *Int. Arch. Photogramm. Remote Sens* 37 (2008), 957–962.
[20] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.
[21] Jakob Engel, Thomas Schöps, and Daniel Cremers. 2014. LSD-SLAM: Large-scale direct monocular Slam. In *European conference on computer vision*. Springer, 834–849.
[22] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. 2015. On-manifold preintegration theory for fast and accurate visual-inertial navigation. *IEEE Transactions on Robotics* (2015), 1–18.
[23] Udo Frese. 2005. A proof for the approximate sparsity of Slam information matrices. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. IEEE, 329–335.
[24] Yiming Gan, Bo Yu, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang, and Yuhao Zhu. 2021. Eudoxus: Characterizing and Accelerating Localization in Autonomous Machines. In *International Symposium on High Performance Computer Architecture (HPCA)*.
[25] Quentin Gautier, Alric Althoff, and Ryan Kastner. 2019. FPGA architectures for real-time dense SLAM. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 83–90.
[26] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for autonomous driving? The KITTI vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition*. IEEE, 3354–3361.
[27] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. 2020. Openvins: A research platform for visual-inertial estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 4666–4672.
[28] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. 2020. OpenVINS: A Research Platform for Visual-Inertial Estimation. In *Proc. of the IEEE International Conference on Robotics and Automation*. Paris, France.
[29] Nima Shoghi Ghalehshahi, Ramyad Hadidi, and Hyesoon Kim. 2019. Slam Performance on Embedded Robots. In *Student Research Competition at Embedded System Week (SRC ESWEEK)*.
[30] Tianyu Gu, Jarrod Snider, John M Dolan, and Jin-woo Lee. 2013. Focused trajectory planning for autonomous on-road driving. In *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 547–552.
[31] Ramyad Hadidi, Bahar Asgari, Sam Jijina, Adriana Amyette, Nima Shoghi, and Hyesoon Kim. 2021. Quantifying the Design-Space Tradeoffs in Autonomous Drones. In *In Proc. of ASPLOS*.
[32] Ramyad Hadidi, Jiashen Cao, ML Merck, Arthur Siqueira, Qiusen Huang, Abhijeet Saraha, Chunjun Jia, Bingyao Wang, Dongsuk Lim, Lixing Liu, et al. 2019. Understanding the power consumption of executing deep neural networks on a distributed robot system. In *Algorithms and Architectures for Learning in-the-Loop Systems in Autonomous Flight, International Conference on Robotics and Automation (ICRA)*, Vol. 2019.

[33] Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael S Ryoo, and Hyesoon Kim. 2018. Distributed perception by collaborative robots. *IEEE Robotics and Automation Letters* 3, 4 (2018), 3709–3716.

[34] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines. *Proceedings of the 33rd International Conference on Computer Graphics and Interactive Techniques*.

[35] Leopoldo Jetto, Sauro Longhi, and Giuseppe Venturini. 1999. Development and experimental validation of an adaptive extended Kalman filter for the localization of mobile robots. *IEEE Transactions on Robotics and Automation* 15, 2 (1999), 219–229.

[36] Daniel S Katz and Raphael R Some. 2003. NASA advances robotic space exploration. *Computer* 36, 1 (2003), 52–61.

[37] Bryan Klingner, David Martin, and James Roseborough. 2013. Street view motion-from-structure-from-motion. In *International Conference on Computer Vision*. IEEE, 953–960.

[38] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[39] George Lentaris, Ioannis Stamoulias, Dimitrios Soudris, and Manolis Lourakis. 2015. HW/SW codesign and FPGA acceleration of visual odometry algorithms for rover navigation on Mars. *IEEE Transactions on Circuits and Systems for Video Technology* 26, 8 (2015), 1563–1577.

[40] Mingyang Li and Anastasios I Mourikis. 2012. Improving the accuracy of EKF-based visual-inertial odometry. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 828–835.

[41] Mingyang Li and Anastasios I Mourikis. 2013. High-precision, consistent EKF-based visual-inertial odometry. *The International Journal of Robotics Research* 32, 6 (2013), 690–711.

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

[43] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. 2018. Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[44] Shuang Liang, Changcheng Tang, Xuefei Ning, Shulin Zeng, Jincheng Yu, Yu Wang, Kaiyuan Guo, Diange Yang, Tianyi Lu, and Huazhong Yang. 2021. Efficient Computing Platform Design for Autonomous Driving Systems. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 734–741.

[45] Q. Liu, S. Qin, B. Yu, J. Tang, and S. Liu. 2020. π-BA: Bundle Adjustment Hardware Accelerator Based on Distribution of 3D-Point Observations. *IEEE Trans. Comput.* 69, 07 (2020), 1083–1095.

[46] Runze Liu, Jianlei Yang, Yiran Chen, and Weisheng Zhao. 2019. eSLAM: An energy-efficient accelerator for real-time ORB-SLAM on FPGA platform. In *Design Automation Conference*. 1–6.

[47] Johan Lofberg. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. In *International conference on robotics and automation*. IEEE, 284–289.

[48] Jun Luo, Qijun Huang, Sheng Chang, Xiaoying Song, and Yun Shang. 2013. High throughput Cholesky decomposition based on FPGA. In *International Congress on Image and Signal Processing (CISP)*, Vol. 3. IEEE, 1649–1653.

[49] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 14–26.

[50] Mark Maimone, Jeffrey Biesiadecki, Edward Tunstel, Yang Cheng, and Chris Leger. 2006. Surface navigation and mobility intelligence on the Mars exploration rovers. *Intelligence for space robotics* (2006), 45–69.

[51] Guoqiang Mao, Sam Drake, and Brian DO Anderson. 2007. Design of an extended Kalman filter for uav localization. In *2007 Information, Decision and Control*. IEEE, 224–229.

[52] Eric Marchand, Hideaki Uchiyama, and Fabien Spindler. 2015. Pose estimation for augmented reality: a hands-on survey. *IEEE transactions on visualization and computer graphics* 22, 12 (2015), 2633–2651.

[53] Luca Marchetti, Giorgio Grisetti, and Luca Iocchi. 2006. A comparative analysis of particle filter based localization methods. In *Robot Soccer World Cup*. Springer, 442–449.

[54] Donald W Marquardt. 1963. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics* 11, 2 (1963), 431–441.

[55] Matthew L Merck, Bingyao Wang, Lixing Liu, Chunjun Jia, Arthur Siqueira, Qiusen Huang, Abhijeet Saraha, Dongsuk Lim, Jiashen Cao, Ramyad Hadidi, et al. 2019. Characterizing the execution of deep neural networks on collaborative robots and edge devices. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 1–6.

[56] Etienne Mouragnon, Maxime Lhuillier, Michel Dhome, Fabien Dekeyser, and Patrick Sayd. 2006. Real time localization and 3d reconstruction. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 1. IEEE, 363–370.

[57] Anastasios I Mourikis and Stergios I Roumeliotis. 2007. A multi-state constraint Kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, 3565–3572.

[58] Raul Mur-Artal and Juan D Tardós. 2017. Orb-slam2: An open-source Slam system for monocular, stereo, and RGB-D cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262.

[59] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. 2016. The microarchitecture of a real-time robot motion planning accelerator. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 45.

[60] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J Sorin, and George Konidaris. 2016. Robot Motion Planning on a Chip.. In *Robotics: Science and Systems*.

[61] Sabrina M Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. 2021. Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 674—-686.

[62] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. *arXiv preprint arXiv:2102.09713* (2021).

[63] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices* 48, 6 (2013), 495–506.

[64] Johannes Pöschmann, Tim Pfeifer, and Peter Protzel. 2020. Factor Graph based 3D Multi-Object Tracking in Point Clouds. *arXiv preprint arXiv:2008.05309* (2020).

[65] Albert Pumarola, Alexander Vakhitov, Antonio Agudo, Alberto Sanfeliu, and Francese Moreno-Noguer. 2017. PL-SLAM: Real-time monocular visual Slam with points and lines. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 4503–4508.

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

[67] Eric Royer, Maxime Lhuillier, Michel Dhome, and Jean-Marc Lavest. 2007. Monocular vision for mobile robot localization and autonomous navigation. *International Journal of Computer Vision* 74, 3 (2007), 237–260.

[68] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. 2018. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 479–490.

[69] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 940–953.

[70] Gabe Sibley. 2006. Sliding window filters for SLAM. *Technical Report CRES-06–004, University of Southern California, Center for Robotics and Embedded Systems* (2006).

[71] Gabe Sibley, Larry Matthies, and Gaurav Sukhatme. 2010. Sliding window filter with application to planetary landing. *Journal of Field Robotics* 27, 5 (2010), 587–608.

[72] Hauke Strasdat, José MM Montiel, and Andrew J Davison. 2012. Visual SLAM: why filter? *Image and Vision Computing* 30, 2 (2012), 65–77.

[73] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. 2019. 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 (2019), 1106–1119.

[74] Ke Sun, Kartik Mohta, Bernd Pfrommer, Michael Watterson, Sikang Liu, Yash Mulgaonkar, Camillo J Taylor, and Vijay Kumar. 2018. Robust stereo visual inertial odometry for fast autonomous flight. *IEEE Robotics and Automation Letters* 3, 2 (2018), 965–972.

[75] Rongdi Sun, Peilin Liu, Jianwei Xue, Shiyu Yang, Jiuchao Qian, and Rendong Ying. 2020. BAX: A Bundle Adjustment Accelerator With Decoupled Access/Execute Architecture for Visual Odometry. *IEEE Access* 8 (2020), 75530–75542.

[76] Niko Sünderhauf, Kurt Konolige, Simon Lacroix, and Peter Protzel. 2006. Visual odometry using sparse bundle adjustment on an autonomous outdoor vehicle. In *Autonome Mobile Systeme 2005*. Springer, 157–163.

[77] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. 1999. Bundle adjustment modern synthesis. In *International workshop on vision algorithms*. Springer, 298–372.

[78] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. 2020. A survey of Fpga-based robotic computing. *arXiv preprint arXiv:2009.06034* (2020).

[79] Jonathan Weisz, Yipeng Huang, Florian Lier, Simha Sethumadhavan, and Peter Allen. 2016. Robobench: Towards sustainable robotics system benchmarking. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3383–3389.

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

[81] Max Willsey, Vincent T Lee, Alvin Cheung, Rastislav Bodik, and Luis Ceze. 2018. Iterative search for reconfigurable accelerator blocks with a compiler in the loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2018), 407–418.

[82] Shuo Yang, Gerry Chen, Yetong Zhang, Frank Dellaert, and Howie Choset. 2020. Equality Constrained Linear Optimal Control With Factor Graphs. *arXiv preprint arXiv:2011.01360* (2020).

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

[84] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 1067–1081.

[85] Fuzhen Zhang. 2006. *The Schur complement and its applications*. Vol. 4. Springer Science & Business Media.

[86] Zhe Zhang, Shaoshan Liu, Grace Tsai, Hongbing Hu, Chen-Chi Chu, and Feng Zheng. 2018. PIRVS: An advanced visual-inertial Slam system with flexible sensor fusion and hardware co-design. In *International Conference on Robotics and Automation (ICRA)*. IEEE, 1–7.

[87] Zhengdong Zhang, Amr Suleiman, Luca Carlone, Vivienne Sze, and Sertac Karaman. [n.d.]. Visual-Inertial Odometry on Chip: An Algorithm-and-Hardware Co-design Approach: Supplementary Material. http://web.mit.edu/sze/www/navion/2017_rss_navion_supplementary.pdf

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

[89] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. 2020. Driving Scenario Perception-Aware Computing System Design in Autonomous Vehicles. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 88–95.