

# PISCES: Power-Aware Implementation of SLAM by Customizing Efficient Sparse Algebra

Bahar Asgari, Ramyad Hadidi, Nima Shoghi Ghalesahi, Hyesoon Kim  
 Georgia Institute of Technology, Atlanta, GA  
 {bahar.asgari, rhadidi, nimash, hyesoon.kim}@gatech.edu

**Abstract**—A key real-time task in autonomous systems is simultaneous localization and mapping (SLAM). Although prior work has proposed hardware accelerators to process SLAM in real time, they paid less attention to power consumption. To be more power-efficient, we propose *Pisces*, which co-optimizes power consumption and latency by exploiting sparsity, a key characteristic of SLAM missed in prior work. By orchestrating sparse data, *Pisces* aligns correlated data and enables deterministic, one-time, and parallel accesses to the on-chip memory. Therefore, *Pisces* (i) eliminates unnecessary memory accesses and (ii) enables pipelined and parallel processing. Our FPGA implementation shows that *Pisces* consumes  $2.5\times$  less power and executes SLAM  $7.4\times$  faster than the state of the art.

**Index Terms**—SLAM, Robotics, Autonomous Systems, Resource Constraint, Sparse Algebra, Power Consumption, FPGA.

## I. INTRODUCTION

The mobility and navigation of autonomous systems such as self-driving vehicles, robots, and drones rely on simultaneous localization and mapping (SLAM). Similarly, SLAM is crucial in the odometry of virtual and augmented reality (VR/AR). To track the location of the agent<sup>1</sup> within a map, SLAM constantly processes the inputs from sensors that periodically scan the environment. To accurately run SLAM, researchers have proposed several algorithms [1]–[5], the key building blocks of which are compute-intensive matrix algebra such as multiplication, transpose, and inversion. Additionally, with continued advancement in sensor technologies with high scanning frequency, SLAM performance is becoming a bottleneck for a faster and more accurate navigation/odometry in autonomous systems. With a limited power budget in autonomous systems, performing the compute-intensive SLAM in real time is a key challenge. This is why several recent studies have accelerated SLAM on hardware [6]–[12].

Although the prior hardware implementations of SLAM have gained performance improvements, they did not particularly aim to consume power most efficiently. The first reason is that none of them consider the sparse structure of the matrix operation in SLAM, which results in inefficient accesses to on-chip memory and high power consumption. Moreover, they often paid less attention to the high ratio of data exchange between a sequence of functions, which potentially consumes high power if not implemented appropriately. For instance, accelerating only the bottleneck-prune functions has been proposed in prior work [6], [7], [9], which is necessary to reduce latency but does not consume power efficiently. The reason is that frequently transferring intermediate data between

the functions accelerated in the hardware and those executed in a host incur extra accesses to on-chip memory and cause high power consumption and outweigh the performance benefits. In other studies, accelerating the preprocessing (i.e., feature extraction) has been proposed [10], [11], which might not be sufficient in achieving energy efficiency and low latency.

We accelerate the entire SLAM algorithm while considering power consumption together with real timeliness. Our main observation is that the computations of SLAM are *sparse* and capture a deterministic structure of *fixed-sized*, small, and dense matrix algebra. We exploit such sparsity, a fundamental attribute of SLAM missed by prior work, to accelerate SLAM. To realize our idea, we propose a power-aware implementation of SLAM by customizing efficient sparse algebra, *Pisces*<sup>2</sup>, which makes the following contributions:

- *Pisces* aligns the correlated dense blocks of data and maps them to adjacent addresses of on-chip memory to enable direct, deterministic, and parallel accesses.
- *Pisces* transforms the sparse matrix algebra to a sequence of fixed-size dense matrix algebra and implements them in a pipelined computation engine, which reads the operands from on-chip memory only once and performs all required operations on intermediate data before writing them back to the memory.

By making the preceding contributions, *Pisces* not only reduces power consumption by eliminating unnecessary accesses to on-chip memory but also improves performance and guarantees real timeliness by enabling pipelined and concurrent processing. For evaluation, we implement *Pisces* and the state-of-the-art peers using Xilinx Vivado high-level synthesis (HLS) tool. We implement them on a ZYNQ XC7Z020 FPGA. Our results show that *Pisces* consumes  $2.5\times$  less power and executes SLAM  $7.4\times$  faster than the state of the art.

## II. SLAM OVERVIEW & RELATED WORK

Since the '90s, several SLAM algorithms have been proposed that are categorized as follows: (i) The direct methods, which use the sensor inputs (e.g., RGB-D camera) to create and process dense maps (e.g., dense visual SLAM [1]); (ii) The indirect feature-based methods, which use a set of features extracted from the sensor inputs rather than the images themselves (e.g., extended Kalman filter (EKF) [2] and oriented-fast and rotated-brief (ORB) [3], [13] SLAMs). The main difference between the EKF and ORB methods is their accuracy. To

<sup>1</sup>Agent refers to a vehicle, robot, drone, or a smart system running AR/VR.

<sup>2</sup>*Pisces* is a constellation including eighteen main stars.

more accurately update a map, the ORB-SLAM uses bundle adjustment (BA) and loop closure optimizations [13], [14], at the expense of performance degradation. As a result, the lightweight EKF could be a more suitable option as long as simplicity outweighs accuracy; (iii) The semi-direct and semi-dense methods that borrow benefits from both of the other categories (e.g., semi direct [4] and LSD [5] SLAM).

To meet real-time constraints, the traditional software implementations of the aforementioned methods on a general-purpose microprocessor (e.g., those used in robots and drones) had not been effective. As a result, hardware implementations such as  $\pi$ -SoC [15] and HERO [16] have focused on optimizing the system performance by offloading the bottleneck-prune parts from CPU to FPGA. To be more impactful, some studies more specifically focused on tailoring the architecture or microarchitecture. The architecture-focused studies have explored (i) FPGA implementations for feature-based SLAM including EKF [6]–[9], ORB [10], [11], LSD [17]–[19], and dense-direct SLAMs [20]; and (ii) ASIC implementations for LSD, ORB [12], and visual internal odometry (VIO) [21].

Microarchitecture-focused studies that are most relevant to our work have mainly focused on feature-based EKF and ORB SLAMs [6]–[12] because of their simplicity. For instance, accelerating the matrix algebra used in EKF algorithm by using one dimensional [6], [7] or Faddeev [9] systolic arrays have been proposed. For ORB, accelerating feature extraction/matching by orchestrating the accesses to features by either using hardware techniques such as synchronized two-stages shifting line buffers [10] or generating rotationally symmetric patterns [11] have been proposed. In this paper, we focus on accelerating EKF and ORB SLAM. Unlike prior work, we target co-optimizing power consumption and latency. To do so, we study and deploy the sparsity feature of SLAM computations that have often been missed in prior work.

### III. CHALLENGES & MOTIVATION

Compared to software implementations, the recently proposed hardware accelerators have been able to improve the performance and power consumption of SLAM. However, the key challenge is that the two following aspects of SLAM that create a performance bottleneck and increase power consumption have remained unstudied: (i) The random accesses to on-chip memory for retrieving correlated data and (ii) The high data-reuse rate of compute-intensive matrix operations of SLAM that results in accesses to on-chip memory. Our key observation to solve this challenge is that since the building blocks of SLAM consist of *sparse* computations, they cause the first aspect and worsen the second one. As a result, we *take advantage of sparsity* to improve the power consumption and performance of SLAM in concert. Before explaining our proposed solution in Section IV, we discuss why sparsity exists in commonly used feature-based SLAM methods.

#### The Sparse Algebra in SLAM:

The sparse matrix operations of SLAM are *structured* and hence create opportunities for optimization. To clarify, we briefly review their main sparse computations in the following.

**EKF** is a feature-based SLAM that has a map consisting of the state of the agent  $(x, y, \theta)$  and the surrounding landmarks  $(x, y)$ , stacked together in a vector  $x_{1 \times N}$  ( $N = 2 \times L + 3$ ,  $L$ : # landmarks). Since the map is modeled by Gaussian variables, it is denoted by a mean vector  $\bar{x}$  and a covariance matrix  $P$  (Figure 1a) that have to be kept updated constantly while the agent moves during predict and update phases, which take 16.56% and 83.4% of the total execution time, respectively:

(1) *Predict*: Based on the prior state  $\bar{x}^{(t-1)}$  and the movement of the agent, the current state  $\bar{x}^{(t)}$  of the agent and the related elements of the covariances matrix  $P^{(t)}$  are predicted:

$$\bar{x}^{(t)} = f(\bar{x}^{(t-1)}, u) \quad (1)$$

$$P^{(t)} = F_x P^{(t-1)} F_x^T + F_q Q F_q^T \quad (2)$$

in which  $f$  is the prediction function,  $u$  is the control vector,  $q$  is the motion noise,  $Q$  is the covariance matrix of  $q$ , and  $F_x$  and  $F_q$  are Jacobian matrices of the motion model with respect to their parameters. Since during this phase, only the agent moves, the largest part of the map remains invariant. Therefore, the Jacobian matrices are *sparse* and result in modifying only the following parts of the  $\bar{x}^{(t)}$  and  $P^{(t)}$ : the state mean of the agent, the covariance of the agent ( $P_{a,a}$ ), and the cross-variances between the agent and the other landmarks ( $P_{a,l}$  and  $P_{l,a}$ ), as shown in Figure 1b.

(2) *Update*: The surrounding landmarks are observed and upon each of the observed landmarks, the covariance matrix of innovation  $Z_{2 \times 2}$  and the Kalman gain  $K_{N \times 2}$  are calculated. Then, the  $\bar{x}^{(t)}$  and  $P_{N \times N}^{(t)}$  are updated:

$$Z = H P^{(t)} H^T + V \quad (3)$$

$$K = P^{(t)} H^T Z^{-1} \quad (4)$$

$$\bar{x}^{(t)} = \bar{x}^{(t)} + K(r - h(\bar{x}^{(t-1)})) \quad (5)$$

$$P^{(t)} = P^{(t)} - K Z K^T \quad (6)$$

in which  $H_{2 \times N}$  is the Jacobian matrix of the observation model,  $V_{2 \times 2}$  is the covariance matrix of observation noise,  $r_{2 \times 1}$  is the observation vector, and  $h$  is the observation function. At each iteration of this phase, Equations 3 to 6 process only one landmark, the observation of which is independent of the other landmarks. As a result, the Jacobian matrix  $H$  is also sparse and updates those elements of  $\bar{x}^{(t)}$  and  $P^{(t)}$  that involve the state of the agent, the state of the concerned landmark, their covariances, and their cross-variances (Figure 1c).

**ORB** is the other feature-based SLAM algorithm, the computations of which engage structured sparse matrices. Here, we focus on BA [13], [14], an optimization in ORB that

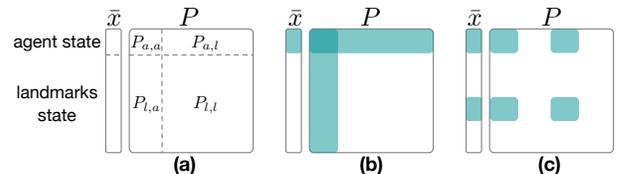


Fig. 1. **Sparsity in EKF algorithm:** (a) The state mean of the agent and landmarks ( $\bar{x}$ ) and covariance matrix ( $P$ ) including the co-variances ( $P_{a,a}$  for agent and  $P_{l,l}$  for landmarks) and cross-variances of agent and landmark with respect to each other ( $P_{a,l}$  and  $P_{l,a}$ ); and the updated parts of the map during the (b) predict phase and (c) update phase of EKF algorithm.

potentially causes a performance bottleneck. For instance, based on our experiments on a Quad-core Cortex-A72 (setup in Section V), the local and global BAs consume 9.7% and 89.23% of total time of ORB SLAM. BA is typically formulated as a non-linear least squares problem that is usually solved by using the Levenberg-Marquardt (LM) algorithm. In general, if  $x$  is a vector of variables, and  $f(x)$  a function of  $x$ , LM formulates the optimization problem as:

$$\min_x \frac{1}{2} \|f(x)\|^2 \quad (7)$$

A general strategy for solving non-linear optimizations is to solve a sequence of approximations to determine a correction  $\Delta x$  to the vector  $x$ , the computational complexity of which can be reduced by using Schur's complement trick, which transforms solving Equation 7 to solving the following equation:

$$H_\mu(x)\Delta x = -g(x) \quad (8)$$

in which  $g$  is the gradient vector, and  $H_\mu$  is the regularized Hessian matrix comprised of two block diagonal *sparse* matrices  $X$  and  $Y$ , and a general block *sparse* matrix  $Z$  the block of which is fixed-size and small:

$$H_\mu = \begin{bmatrix} X & Z \\ Z^T & Y \end{bmatrix} \quad (9)$$

As Figure 1 and Equation 9 show, EKF and ORB capture common features of sparsity: (i) their sparse matrix computations comprise a sequence of matrix operations on sparse matrix operands that capture fixed-size, small, and dense blocks of data; and (ii) the *correlated* dense blocks corresponding to a single sparse matrix operation are scattered over *deterministic related locations* of the original matrix.

#### IV. PISCES

This section explains the key insights of Pisces and introduces its microarchitecture and system overview.

##### A. Key Insights

To *efficiently* improve SLAM performance, Pisces leverages the structure of sparse computations to improve the locality of accesses to the on-chip memory, optimize the matrix operations (i.e., the small fixed-size matrix operations on dense operands), and reduce accesses to the on-chip memory. Before focusing on details, we explain the fundamental insights on which we build our system. Our first insight is to align the correlated data (i.e., the dense blocks of data that are processed together) and map them to adjacent locations of the on-chip memory.<sup>1</sup> Such a data ordering/mapping is clarified by the following example in the update phase of the EKF SLAM. As Equations 3 and 5 show, matrices  $P$  and  $H$  and vector  $x^{(t)}$  (together called the *map*) are the inputs to the update phase and have the structures shown in Figure 2a. The size of each block is  $2 \times 2$ , which depends only on the coordinate system (e.g., 2 represents  $x$  and  $y$  in a Cartesian coordinate system). The colored blocks correspond only to one specific iteration of processing landmarks, and the gray blocks (i.e., the covariance

<sup>1</sup>For the rest of the paper, the on-chip memory and BRAM are interchangeable. This is because although, in this paper, we target an FPGA implementation, the insights are also applicable to an ASIC design.

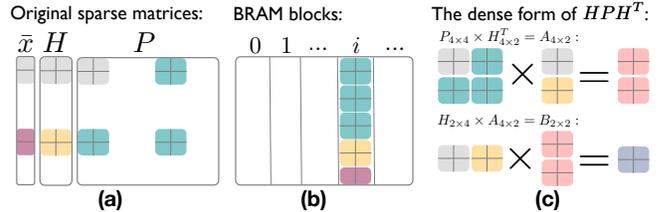


Fig. 2. **The key insight to accelerate the update phase of EKF:** (a) An example of the parts of the maps and the Jacobian matrix required for executing the update phase of EKF for updating the map upon observing one landmark. (b) Mapping the data corresponding to the observed landmark to a single block of BRAM. (c) Performing an example of sparse matrix multiplication as two small dense matrix multiplications.

of the agent,  $P_{a,a}$ ) are common to all of them. To access only the colored block and to access them together *directly*, Pisces maps them to a single block of BRAM (Figure 2b). Our second insight is to implement the entire chain of sparse computations (e.g., Equations 3 to 6) as a sequence of dense matrix operations that read a block of BRAM only once and apply all required processes before writing it back.

Based on the insights, we implement *data reuse* by flowing data between the compute units through registers rather than through BRAM blocks. Such an implementation is more effective in achieving high power efficiency but is not always feasible because of the non-deterministic nature of accesses. However, the following crucial characteristic of matrix algebra in SLAM makes it possible. First, we put the  $2 \times 2$  blocks of data together, regardless of their original location in matrix  $P$ , and preform a matrix operation on the new fixed dense matrix operands without preserving the locations of the blocks. For instance, the  $HPH^T$  of Equation 3 is transformed to two fixed matrix multiplications shown in Figure 2c.

The second and more important characteristic of sparse SLAM is that *the sparsity and the location of dense blocks propagates through the equations*. Thus, without needing to preserve the location of the dense blocks, we continue performing the required dense matrix operations in a sequence by passing the intermediate results through operations and still maintain the functionality. The third characteristic is that the intermediate data flowing through the dense matrix operations are small and do not need addressable accesses to memory. Therefore, they can be implemented either by using registers or by just being passed in a combinational logic, whichever better suits the design choices such as clock frequency.

##### B. Microarchitecture

The microarchitecture of Pisces consists of modules of dense fixed-size matrix operations that are placed together in two levels of pipeline in a specific configuration to accelerate a particular SLAM algorithm.

*Building Blocks:* The matrix algebra required by EKF and ORB SLAM include multiplications, transpose, inverse, sum, and subtraction. Because of the  $2 \times 2$  size of the blocks captured in the original sparse matrices and the deterministic distribution patterns of such blocks (discussed in Section III), the dimensions of the operands are always either 2 or 4, which depends only on the coordinate system. Therefore, we categorize the operands based on their sizes into (i) small,

TABLE I  
THE LATENCY OF THE MODULES OF DENSE MATRIX OPERATIONS.

Matrix Op.	Transpose	Mult. LM*	Mult. MM*	Mult SM/MS*	Sub/Sum
Latency( $\mu$ s)	0.06	0.66	0.33	0.58	0.09

\*LM: Large-Medium, MM: Medium-Medium, SM: Small-Medium, MS: Medium-Small.

$2 \times 2$ , (ii) medium,  $2 \times 4$  or  $4 \times 2$ , and (iii) large,  $4 \times 4$ . The implementation and optimization of matrix operations on such operations are straightforward. For instance, regardless of the complexity of matrix inversion, it can be implemented by a small full combinational logic. For Pisces, we implement the matrix operation on FPGA (see Section V for details) and list their latencies in Table I. Note that in the specific case of EKF and the BA of ORB, we only need transpose on medium-size matrices and inverse on small-size matrices. Besides, for all the multiplications, only the most inner loop is parallelized.

*Pipeline Overview & Configurations:* Pisces implements two levels of pipelines in its main microarchitecture, as shown in Figure 3. The first level is the outer pipeline that enables streaming the map from a source (whether from host CPU, or another module in FPGA). The three stages of this pipeline are (i) input stage, streaming in the input map to the input buffers, (ii) process stage, processing the map and writing the updated map to the output buffer, and (iii) output stage, streaming out the content of the output buffer. The buffers of this pipeline are implemented in BRAM. For EKF, the BRAM blocks contain the colored blocks shown in Figure 2b. The gray blocks are sent separately to the hardware accelerator (through registers) and are not shown in Figure 3.

The second level, inner pipeline, breaks down the process stage into smaller stages, each implemented by one (or a combination of more) dense matrix operation listed in Table I. In this level of pipeline, the number of stages and the operation of each stage are defined by the algorithm. For instance, as the table in Figure 3 illustrates, the update function of EKF and the BA of ORB require six and four stages, respectively. As shown, for balancing the stages, transpose, inverse, and sum/sub, which are faster (see Table I), are combined with the multiplications. For EKF, the sequence of operations are derived from Equations 3 to 6. For BA of ORB, the details of all equations are not explained for the sake of brevity, but readers are encouraged to read the references [3], [13].

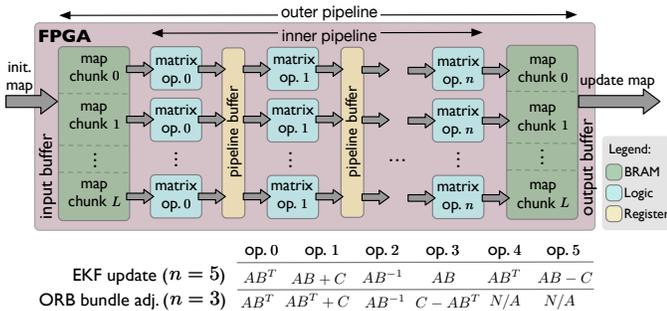


Fig. 3. The microarchitecture of Pisces: Streaming the map chunks, processing them concurrently through independent pipelined matrix operations, and streaming out the updated chunks of map ( $L$ : # of landmarks).

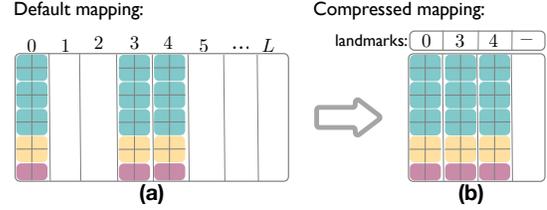


Fig. 4. The compression scheme: (a) The one-on-one mapping of observed landmarks to BRAM blocks. (b) Mapping the observed landmarks to adjacent BRAM blocks and saving their indices separately.

### C. Design Optimization

Dedicating BRAM blocks to buffer the entire map in the FPGA, and one-on-one mapping of blocks to landmarks (as shown in Figure 4a) is not the most efficient way of utilizing the on-chip memory. More specifically, because, over time, the agent observes new landmarks and stops observing some of the old ones. Therefore, we need to dynamically replace old landmarks with new ones. As a result, we exploit a simple compression scheme shown in Figure 4b. Since the operations can be performed regardless of the absolute location of a map chunk in the original matrix, we simply compress them and perform the operations on a given number of chunks. For instance, as shown in Figure 4b, we choose to stream groups of four landmarks and process four of them in parallel to sustain a moderate resource utilization (Table II). However, since the design is modular, larger group sizes and more levels of parallelism can be easily configured (Section VI).

### D. System Architecture

The explained microarchitecture of Pisces (Figure 3) is designed to accelerate the entire time-consuming iterative processes of SLAM (e.g., processing the landmarks in the update phase of EKF or the BA of ORB). When we implement Pisces on an SoC system, if the real-time constraint is not hard, the non-iterative processes (e.g., the predict phase of EKF) can be executed in the CPU. In such a system, as shown in Figure 5, the CPU generates the list of landmarks and streams them in groups (e.g., groups of four chunks of the map) to the FPGA. Otherwise, in a hard real-time system, as a design choice, all the tasks should be pushed to the FPGA. Such tasks include handling the sensor inputs and generating the list of observed landmarks, which consists of a combination of the same matrix operations listed in Table I.

## V. EVALUATION METHODOLOGY

We implement Pisces and the baselines using Xilinx Vivado HLS. We use relevant `#pragma` as hints to describe our desired microarchitectures in C++. For instance, the two levels of pipelines (Figure 3) are implemented using `dataflow` pragma. To create four instances of the inner pipeline, we use `unroll` pragma and enable parallel accesses to BRAM buffer

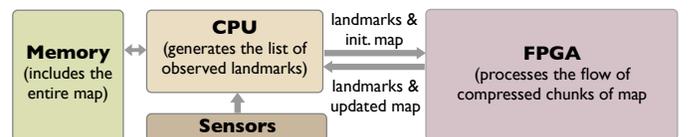


Fig. 5. The system overview of Pisces.

TABLE II  
RESOURCE UTILIZATION AND THE TOTAL POWER CONSUMPTION.

	EKF			ORB		Available
	1DSA [7]	FSA [9]	Pisces EKF	eSLAM [11]	Pisces ORB	
BRAM(Kb)	756	297	252	78	180	2520
LUT	7824	3073	14472	56954	11898	53200
FF	4223	5176	16686	67809	12178	106400
DSP	32	2	75	111	114	220
Power(W)	1.302	0.986	0.384	1.936	0.292	N/A

using `array_partition` pragma. We target the SoC system of the PYNQ-z1 board. Therefore, we synthesize and implement Pisces and the baselines on its FPGA, a ZYNQ XC7Z020. We present the post-implementation resource utilization, power consumption, and latency, reported by Vivado. Inputs and outputs of the accelerators are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. To study the performance of EKF, we emulate various environments with a various number of landmarks. Such a methodology helps us to explore the scalability feature of EKF, which is known to be the drawback of EKF. For ORB, we evaluated Pisces and the baseline using the EuRoC dataset.

We compare Pisces against two state-of-the-art hardware accelerators for SLAM, including a one-dimensional systolic array (1DSA) [7] and the Faddeev systolic array (FSA) [9] for EKF. For ORB SLAM, we implement eSLAM [11] as a complimentary accelerator to explore the benefits of using Pisces and eSLAM together for further improvement. We also implement the EKF and ORB SLAMs on a Raspberry Pi 4 board not only to study the performance of executing SLAM on Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) SoC @1.5GHz with 4GB SDRAM, but also to validate the functionality and accuracy of FPGA implementation.

## VI. RESULTS

This section evaluates the resource utilization, power consumption, and latency of Pisces compared with the baselines.

### A. Resource Utilization & Power Consumption

The resource utilization of Pisces (for both EKF and ORB configurations) and the baseline hardware accelerators, as well as the available resources of the target FPGA, are listed in Table II. As the numbers show, for EKF SLAM, Pisces employs fewer or an equal number of BRAM blocks while using up to  $4\times$  more LUT and FF as those utilized by 1DSA and FSA. As this comparison suggests, Pisces trades BRAM for FF and LUT to more efficiently consume the power budget. The impact of such a trade-off is  $3.3\times$  and  $2.5\times$  less power consumption compared to 1DSA and FSA, respectively, as is listed in Table II (the total power also includes the activity of clock). The table also lists the resource utilization and power consumption of eSLAM and the ORB configuration of Pisces that accelerate two different parts of ORB. However, in an FPGA with limited available resources, such as our

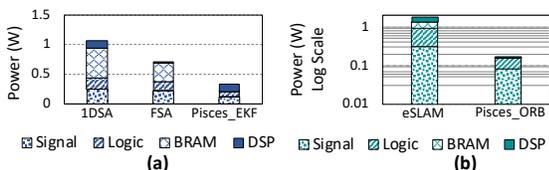


Fig. 6. Power consumption breakdown: (a) EKF, (b) ORB.

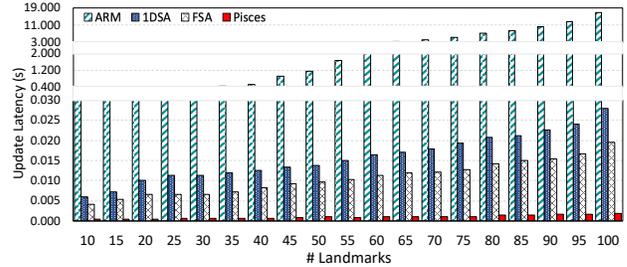


Fig. 7. The latency of update phase of EKF: Comparing Pisces with the prior hardware accelerators and the software implementation on ARM processor when the number of landmarks varies.

target FPGA, implementing both eSLAM and Pisces together exceeds the available LUTs and DSPs. In such a case, we suggest accelerating only the local and global BA, as they are the time-consuming parts. As a point of comparison, running SLAM on the ARM processor of Raspberry Pi 4 consumes approximately  $5W$ .

The breakdown of the power consumption, shown in Figure 6, illustrates the impact of BRAM accesses on the total power consumption. For instance, although FSA and the EKF configuration of Pisces use a similar amount of BRAM, the power consumed by the BRAM in Pisces (i.e.,  $0.009W$ ) is only a tiny fraction of the total power. More specifically, 1DSA mainly targets accelerating Equation 6. Besides the accesses to BRAM risen by each call to Equation 6, to accelerate this equation, 1DSA uses a one-dimensional array of multiply and accumulate (MAC) units and hence frequently writes back the intermediate data to the BRAM. On the other hand, FSA implements the Schur's complements to accelerate the linear equations and matrix inversions of the EKF algorithm. FSA uses connected arrays of processing elements (PEs), in which each PE is responsible for the computation of one row of FSA. Since the sizes of the PEs are fixed, and as they did not leverage sparsity, the FSA operations are performed in tiles. As a result, handling the boundary condition of tiles generates extra access to BRAM.

### B. Latency

**EKF:** First, we compare the latency of the update phase of EKF with the software implementation (on the ARM processor), 1DSA, and FSA. As Figure 7 shows, only when a few (i.e., less than 15) landmarks exist in the environment, is the latency of ARM smaller than  $0.03s$ , which indicates processing the sensor inputs generated by 30 frames per second (fps) rate. Note that a typical urban environment easily has more than 100 trackable landmarks, if not more. For a higher

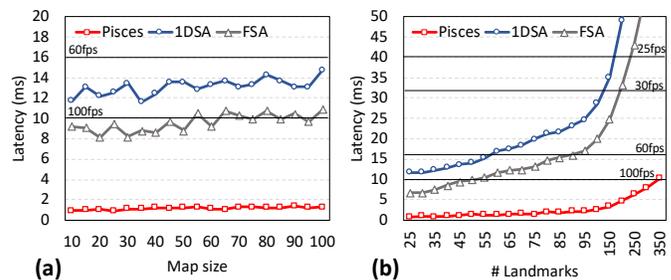


Fig. 8. Total latency of EKF: Comparing Pisces with two prior hardware implementations of EKF SLAM. (a) 50 landmarks scattered over various map sizes. (b) 20 to 350 landmarks scattered over a map of size 100.

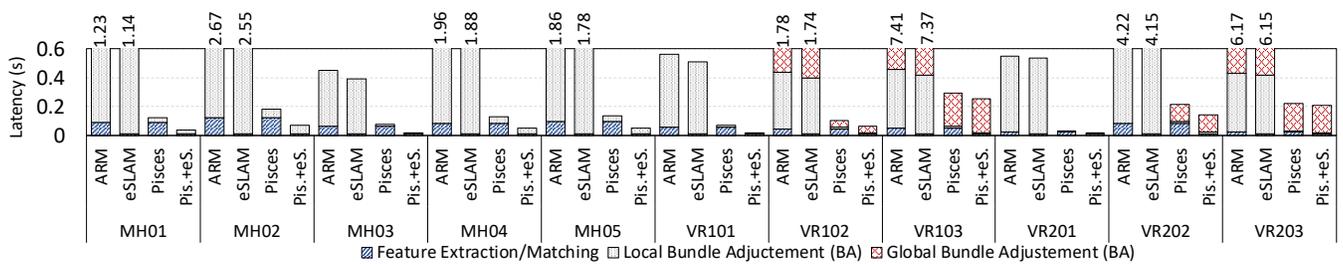


Fig. 9. **Latency breakdown of ORB:** Comparing ARM, eSLAM, Pisces, and a combination of Pisces and eSLAM (i.e., Pis.+ eS.).

number of landmarks, the latency of ARM implementation grows exponentially. The same story occurs for the prior hardware implementations with a high number of landmarks, even though their latencies are an order of magnitude smaller than the software implementation. In fact, we see that the power consumption is not the only concern about the prior hardware accelerators, as we clarify in the following.

Seeking a fair comparison for total EKF latency, similar to Pisces configuration (details in Section IV-C), we apply a four-level parallelism to the prior hardware accelerators. Besides, we assume that all the accelerators execute the first phase (i.e., predict) in FPGA and spend  $2.85\mu\text{s}$  on Equations 1 and 2. Moreover, we assume that, similar to the baselines, Pisces does not benefit from pipelining for processing each chunk of the map. The results of this experiment, Figure 8, show that on average Pisces executes EKF approximately  $11\times$  and  $7.4\times$  faster than 1DSA and FSA, respectively. The speedup of Pisces stems from implementing the sparse operations as a chain of fixed-size dense matrix operations. In particular, executing Equation 6 (i.e., the bottleneck-prune equation) takes  $1.18\mu\text{s}$ , which is approximately  $15\times$  as fast as 1DSA.

As Figure 8a shows, when fewer landmarks (e.g., 50) exist in an environment, regardless of the map size, 1DSA and FSA can meet the real-time constraints defined by a 60 or less fps sensor. However, as Figure 8b shows, as soon as the number of landmarks increases, none of the prior hardware accelerators are fast enough, even if the sensor rate is as low as 25 fps. In a crowded environment (e.g.,  $>350$  landmarks), the four-level concurrent processing of the current Pisces configuration might not deliver an appropriate latency. In such cases, we recommend increasing the level of concurrency. The modular design of Pisces eases such a modification.

**ORB:** Figure 9 presents the latency breakdown of ORB SLAM including the time spent on feature extraction and matching that is accelerated by eSLAM, and the local and global BAs that are accelerated by our work, Pisces. As the figure shows, although eSALM extracts/matches the features  $8\times$  as fast as the software implementation on ARM, it does not have a significant impact on total latency. On the other hand, Pisces significantly reduces the latency of local and global BA. As a result, to meet real-time constraints, we can combine the two approaches if the available resources of the target FPGA allow. Otherwise, we prioritize accelerating BA.

## VII. CONCLUSIONS & FUTURE WORK

This paper proposed Pisces, a new approach to accelerate SLAM. To improve power consumption and latency, Pisces

transformed the sparse matrix operations into a chain of fixed-size dense matrix operations. Pisces reduces the accesses to BRAM by implementing the data exchange between the functions by using registers rather than BRAM accesses.

## ACKNOWLEDGMENT

We gratefully acknowledge the support of NSF CSR 1815047.

## REFERENCES

- [1] C. Kerl, J. Sturm *et al.*, “Dense visual slam for rgb-d cameras,” in *IROS*. IEEE, 2013, pp. 2100–2106.
- [2] M. G. Dissanayake, P. Newman *et al.*, “A solution to the simultaneous localization and map building (slam) problem,” *IEEE Transactions on robotics and automation*, vol. 17, no. 3, pp. 229–241, 2001.
- [3] E. Rublee, V. Rabaud *et al.*, “Orb: An efficient alternative to sift or surf,” in *ICCV*, vol. 11, no. 1. Citeseer, 2011, p. 2.
- [4] J. Engel, J. Sturm *et al.*, “Semi-dense visual odometry for a monocular camera,” in *ICCV*, 2013, pp. 1449–1456.
- [5] J. Engel, T. Schöps *et al.*, “Lsd-slam: Large-scale direct monocular slam,” in *ECCV*. Springer, 2014, pp. 834–849.
- [6] D. T. Tertei, J. Piat *et al.*, “Fpga design and implementation of a matrix multiplier based accelerator for 3d ekf slam,” in *ReConFig*. IEEE, 2014, pp. 1–6.
- [7] D. Tertei, J. Piat *et al.*, “Fpga design of ekf block accelerator for 3d visual slam,” *Computers & Electrical Engineering*, vol. 55, pp. 123–137, 2016.
- [8] M. S. Hanif, M. Bilal *et al.*, “Implementation of an embedded testbed for indoor slam,” in *AICCSA*. IEEE, 2018, pp. 1–8.
- [9] L. de Souza Rosa, A. Dasu *et al.*, “A faddeev systolic array for ekf-slam and its arithmetic data representation impact on fpga,” *Journal of Signal Processing Systems*, vol. 90, no. 3, pp. 357–369, 2018.
- [10] W. Fang, Y. Zhang *et al.*, “Fpga-based orb feature extraction for real-time visual slam,” in *ICFPT*. IEEE, 2017, pp. 275–278.
- [11] R. Liu, J. Yang *et al.*, “eslam: An energy-efficient accelerator for real-time orb-slam on fpga platform,” in *DAC*. ACM, 2019, p. 193.
- [12] H. Chen, Y. Dai *et al.*, “Towards efficient microarchitecture design of simultaneous localization and mapping in augmented reality era,” in *ICCD*. IEEE, 2018, pp. 397–404.
- [13] R. Mur-Artal and J. D. Tardós, “Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras,” *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
- [14] S. Agarwal, N. Snavely *et al.*, “Bundle adjustment in the large,” in *ECCV*. Springer, 2010, pp. 29–42.
- [15] J. Tang, B. Yu *et al.*, “ $\pi$ -soc: Heterogeneous soc architecture for visual inertial slam applications,” in *IROS*. IEEE, 2018, pp. 8302–8307.
- [16] X. Shi, L. Cao *et al.*, “Hero: Accelerating autonomous robotic tasks with fpga,” in *IROS*. IEEE, 2018, pp. 7766–7772.
- [17] M. Gu, K. Guo *et al.*, “An fpga-based real-time simultaneous localization and mapping system,” in *FPT*. IEEE, 2015, pp. 200–203.
- [18] K. Boikos and C.-S. Bouganis, “Semi-dense slam on an fpga soc,” in *FPL*. IEEE, 2016, pp. 1–4.
- [19] K. Boikos and C. S. Bouganis, “A scalable fpga-based architecture for depth estimation in slam,” in *ARC*. Springer, 2019, pp. 181–196.
- [20] Q. Gautier, A. Althoff *et al.*, “Fpga architectures for real-time dense slam,” in *ASAP*, vol. 2160. IEEE, 2019, pp. 83–90.
- [21] A. Suleiman, Z. Zhang *et al.*, “Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones,” *IEEE Journal of Solid-State Circuits*, vol. 54, no. 4, pp. 1106–1119, 2019.