FSLAM: an Efficient and Accurate SLAM Accelerator on SoC FPGAs

Vibhakar Vemulapati
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL, USA
vemulpt2@illinois.edu

Deming Chen
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL, USA
dchen@illinois.edu

2022 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-5336-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICFPT56656.2022.9974562

***Abstract*—Simultaneous Localization and Mapping (SLAM) is one of the main components of autonomous navigation systems. With the increase in popularity of drones, autonomous navigation on low-power systems is seeing widespread application. Most SLAM algorithms are computationally intensive and struggle to run in real-time on embedded devices with reasonable accuracy. ORB-SLAM is an open-sourced feature-based SLAM that achieves high accuracy with reduced computational complexity. We propose an FPGA based ORB-SLAM system, named FSLAM, that accelerates the computationally intensive visual feature extraction and matching on hardware. FSLAM is based on a Zynq-family SoC and runs 8.5x, 1.55x and 1.35x faster compared to an ARM CPU, Intel Desktop CPU, and a state-of-the-art FPGA system respectively, while averaging a 2x improvement in accuracy compared to prior work on FPGA.**

***Index Terms*—Autonomous Navigation, FPGA Accelerator, Computer Vision**

I. Introduction

Simultaneous Localization and Mapping (SLAM) is the problem of constructing a map of an unknown environment, while simultaneously estimating the current pose (position and orientation w.r.t a fixed reference) of the observer. The mapping and pose need to be solved simultaneously because pose estimates from odometry alone are subject to noise and drift, while constructing a map requires a good estimate of your current pose. SLAM is a fundamental problem in autonomous navigation and location estimation with applications in autonomous cars/drones and augmented reality. SLAM has the potential opportunity of replacing or enhancing GPS tracking and navigation in certain applications and environments. GPS systems are not accurate indoors or in big cities, where there are obstructions between the navigation satellites and the observer. Moreover, in the best of conditions, GPS is only accurate to within a meter in the horizontal plane and around 3 meters in the vertical direction. SLAM offers the benefit of not relying on external beacons (satellites) in unreliable environments while potentially being more accurate about the observer's location.

Visual SLAM solves the SLAM problem using visual data for both pose estimation and mapping. The visual data can be either from monocular images, stereo images, or 3-D images. With the rise in popularity in UAVs (unmanned aerial vehicles), visual SLAM algorithms are gaining traction in embedded systems that run in low-power environments, especially if they can perform well with low-quality sensors. Visual SLAM algorithms can be broadly categorized based on two criteria: the density of the map being reconstructed and the type of inputs being tracked. Reconstructed maps can be dense, which reconstruct the entire space, semi-dense, which reconstruct detected edges, and sparse, which only reconstruct features. Features are points of interest (generally corners) in each frame that are tracked across frames as points of reference. Visual SLAM algorithms either attempt to track every pixel in the frame or attempt to track features detected.

One of such visual SLAM algorithms published as an open source library is ORB-SLAM [1] [2] [3]. ORB-SLAM utilizes ORB (Oriented FAST, Rotated Brief) features explained in more detail in Section III-C. It gained significant popularity in the visual SLAM community for implementing a highly accurate monocular SLAM system. Later, the work was extended to utilize stereo images [2], as well as RGB-D images, which at the time was one of the most accurate visual SLAM algorithms publicly available. ORB-SLAM proposes a feature-based sparse visual SLAM, making it relatively less computationally intensive (compared to other SLAM systems). Inspite of that, ORB-SLAM still struggles to run on embedded devices. Table I shows the performance of ORB-SLAM on various platforms when using RGB-D inputs. NVIDIA's Jetson Xavier platform, which sports a large 8-core ARMv8.2 and is marketed towards robotics applications, is only able to achieve 10 frames per second using purely the CPU, and 22fps when utilizing the CPU and the GPU [4]. We also observe that the feature extraction component accounts for 60-70% of the runtime of the algorithm. Feature extraction is an image processing algorithm, which involves many parallel computation patterns, making it well suited for FPGAs.

SLAM acceleration using FPGAs is a difficult problem as there are multiple constraints that need to be met. The system needs to have real-time performance, high accuracy, and be able to fit within an embedded device. There has been some work in accelerating ORB feature extraction by itself, which is evaluated by accuracy of tracking features across frames. Previous works have experimented with hardware optimizations of ORB feature extraction and matching by trading off accuracy for performance. The accuracy of SLAM

TABLE I: Average Performance of ORB SLAM on various platforms. "ORB runtime" indicates the time of the frame spent extracting ORB features.

Device	FPS	Total Frame runtime (ms)	ORB runtime (ms)
Intel i5 Desktop	40	25	16
Intel i5 Laptop	33	30	18
NVIDIA Jetson (CPU)	10	100	70
NVIDIA Jetson (CPU+GPU)	22	45	20
ARM CPU	7	142	100

systems is inherently noisy, and hence, the performance of ORB feature matching does not necessarily correlate with the end performance of visual SLAM. In this work, we propose hardware optimizations to improve the run-time performance on embedded devices. We investigate how these hardware optimizations affect the final accuracy of the SLAM system, while relaxing the accuracy demands of intermediate computations, allowing us to make global optimizations that have been overlooked. The contributions of this work can be summarized as follows:

- FSLAM, an open-sourced implementation of an end-to-end SLAM accelerator that runs on an FPGA SoC.
- A hardware-based accelerator for the ORB feature extraction, which can support multiple image resolutions and multiple levels of image pyramid processing in parallel.
- Data-driven optimizations and approximations such as bit-width pruning and data quantization to ensure resource efficiency, while minimizing impact on the accuracy of the overall system.

The rest of the paper is organized as follows: Section II provides an overview of prior implementations of ORB and ORB-SLAM algorithms in hardware. Section III introduces the ORB-SLAM algorithm. Section IV describes the hardware architecture of our ORB accelerator. Section V highlights the experimental results that guided our optimizations. Section VI summarizes our results and insights in this paper.

II. Related Work

FPGAs have become popular in the embedded device space as they enable real-time performance while maintaining a low energy profile compared to embedded CPUs or GPUs.

The major drawbacks in deploying FPGA accelerators compared to CPUs or GPUs are constrained resources and longer development times. FPGAs have far fewer arithmetic units, memories and logical elements when compared to GPUs. A general strategy for migrating applications to FPGAs involves optimizing the algorithm to fit on resource-constrained devices while maintaining real-time performance. Zhang et al. [5] and Li, et al. [6] propose modifications such as bit-width pruning, data quantization and software-hardware co-design to fit the applications on embedded devices without sacrificing much accuracy.

SLAM algorithms are parallelizable and energy intensive, and hence are well suited to FPGA acceleration. There have been several prior works that have investigated SLAM acceleration on FPGAs. The works can be broadly categorized into two buckets: front-end ORB feature extraction accelerators, and back-end Bundle Adjustment accelerators. Fang, et al. [7] implemented an ORB accelerator to integrate into their visual SLAM flow. They were able to achieve 67 fps on 640x480 images on an FPGA running at 200MHz. They optimized some resources by truncating the width of the intermediate values of computation. Weberruss et al. [8] proposed a multi-scale ORB accelerator. They achieved 72fps on 1920x1080 images, processing the image at 1 pixel/cycle. Tran et al. [9] propose a stream-based ORB accelerator focusing on dynamic power optimizations using dynamic clock gating, and threshold-guided bit-width pruning of intermediate computation values. No performance numbers were reported. Both referenced studies [8], [9] use the Harris Corner Detection algorithm instead of FAST for corner detection. Although Harris Corner is more complex computationally, it provides better accuracy in corner extraction. Qin, et al. [10] propose an FPGA SoC based accelerator to target the Bundle adjustment portion of SLAM. They proposed a hardware/software co-designed accelerator which sped up the Schur Complement computation of the algorithm. Their work achieved a speedup of 1.5x compared to the ARM implementation. The authors further extended on this work in [11] and improved the speedup to 7.5x compared to ARM. Wu, et al. [12] propose an FPGA accelerator for DS-SLAM, a neural network based SLAM [13]. Neural-network based SLAMs have been emerging in the field, as they are more accurate; however, the computational cost is an order of magnitude higher. The authors implement a platform called HERO that is 5x more energy efficient than a desktop system, while running at 5fps. Liu, et al. [14] propose a framework for generating hardware for localization acceleration. They are able to achieve speed-ups upto 8x over Desktop CPUs, while consuming a great amount of resources.

A few works implement end-to-end SLAM accelerators. Boikos, et al. [15] implement the full LSD-SLAM pipeline on a Zynq SOC. LSD-SLAM is a semi-dense direct visual SLAM algorithm [16]. They offload compute-intensive optimization problems onto the FPGA while the CPU takes care of bookkeeping and other minor tasks. They were able to achieve 22fps on 640x480 images. Asgari, et al. [17] proposed an end-to-end ORB-based SLAM system that uses EKF (Extended Kalman filters) as the optimization backend. EKFs have not been used in the robotics community for a few years, as graph-based approaches have taken over the back-end of SLAM. The authors implement a system that is more power efficient and faster than eSLAM [18], but fail to mention accuracy in the paper, so it is difficult to gauge if it performs in real-world scenarios. Liu, et al. [18] implement a visual SLAM system based on ORB features on a Zynq SoC. Feature extraction and matching was offloaded to the FPGA, while the pose optimization, pose estimation and map updating were performed on the CPU. They also proposed a hardware-friendly optimization to BRIEF patterns, which costs fewer FPGA resources, but produces a less accurate system.

We compare the hardware implementations of the above works in more detail in Section IV-E.

III. ORB-SLAM

A. SLAM

SLAM algorithms consist of two parts: map generation and localization. The map generation places points in 3-d space, which can then be used as reference points for localization. Localization estimates the pose of the camera by observing the changes in features across consecutive frames. Modern visual SLAM algorithms use a keyframe based approach where the keyframes contain information about the map points they observe. A new keyframe is generated when either a certain amount of time has passed or a certain number of new points are observed that have not been observed in other keyframes. Localization is performed using the position estimates of features observed in the current frame that are observed in adjacent keyframes using a bundle adjustment algorithm detailed further in Section III-B. Keyframe based approaches use a graph-based method for map point and pose estimation. In contrast, earlier SLAM algorithms used a filter-based approach, where localization and mapping were performed simultaneously. Graph-based SLAM has become more popular recently with works like Grisetti, et al. [19] explaining why it is more accurate.

ORB-SLAM is a sparse feature-based visual SLAM algorithm capable of operating with monocular, stereo, and monocular + depth inputs. The algorithm consists of three main threads that are running in parallel:

- Tracking: Extracts ORB features, performs pose estimation and adds keyframes.
- Local Mapping: Updates the map with new points from keyframe. Performs optimization of local keyframes (Local Bundle Adjustment). Removes redundant keyframes and bad map points.
- Loop Closing: Checks to see if an area has been revisited, allowing the algorithm to re-calibrate the pose and correct any drift accumulated over time. Optimizes all the keyframes (Global Bundle Adjustment) on loop closure. It is an optional feature that greatly enhances accuracy in scenarios where the same location is visited multiple times.

B. Bundle adjustment (BA)

Bundle adjustment(BA) is an algorithm in photogrammetry used to reconstruct an image from multiple viewpoints [20]. A 3-D point is reprojected from the camera's 2-D reference frame into the world coordinates based on multiple observations from different viewpoints. As in Figure 1, the same point on the cuboid is observed from multiple frames, with each observation having inherent uncertainty. Since there are more equations than variables, BA attempts to minimize the reprojection error, across all equations using a non-linear regression. ORB-SLAM has 3 types of BA, each running on independant threads: Motion-only BA, local BA, and global BA.

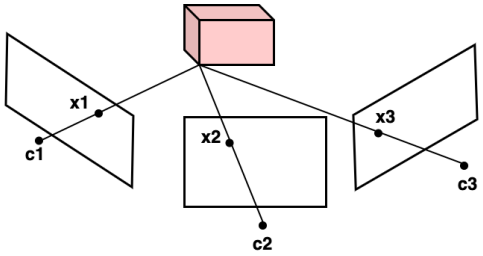

Fig. 1: Bundle adjustment estimation example. The same point on the cuboid is observed from 3 different camera positions $c1$, $c2$, $c3$. The corner is projected onto the 2-d frame at $x1$, $x2$, $x3$.

- Motion only BA optimizes the camera's position and orientation with respect to the features observed in the current frame. It does not update the map points. For given position $t \in (x, y, z)$ and orientation $R \in SO(3)$ (3-D rotation group), we optimize for pose using the following equation, which aims to minimize the error between observed and predicted position of every point in the frame.

$$\{R, t\} = argmin \sum_{i \in \chi} ||x^i - X^i(R, t)||^2$$

  χ is the set of all features of the current frame matched with the map points, with x being a matched feature, and X, the projection of the matched map point into the current frame.
- Local BA optimizes all the map points observed in a given set of keyframes that share observations. It is executed on the addition of a new keyframe.
- Global BA optimizes all the keyframes detected so far (with the exception of the first keyframe). This is done when a loop is detected to correct the observations of all the map points.

C. ORB

ORB [21] is a feature extraction and description algorithm that efficiently identifies features in an image and generates a unique descriptor for each feature in order to identify them in other frames. ORB has been proven to be robust, while maintaining a relatively simple computational pipeline, making it attractive for feature-based sparse SLAM algorithms. ORB (Oriented-FAST and Rotated-BRIEF) can be broken down into two parts: feature extraction (oriented-FAST) and feature description (rotated-BRIEF). oFAST extracts corners and determines their orientation. rBRIEF generates a rotation invariant descriptor. The ORB computation pipeline is shown in Figure 2. The algorithm takes in the image, and outputs the keypoints and their associated descriptors. In the following sections, the ORB algorithm will be described in more detail.

D. Image Pyramid

ORB extracts features from multiple scales of an image: the original image, and multiple downsampled versions of

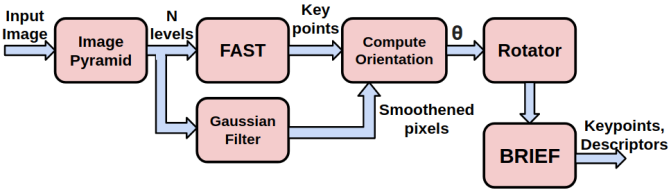

Fig. 2: ORB Pipeline

the original. Feature extraction algorithms are usually implemented with a fixed-size sliding window over the input image. Multiple image scales effectively increase the size of the sliding window, allowing it to detect features, that would otherwise, not be seen in different scales. For instance, large, rounded corners would be missed in larger images, while at smaller scales, they get shrunk down to a detectable size. Image pyramids also enhance the efficiency when the input camera is in motion, where the same object would be detected at multiple scales as the camera moves towards/away from it. The image pyramid used in the ORB-SLAM pipeline consists of 4 levels, with each level 1.2 times smaller than the previous.

E. FAST

FAST examines the local environment of a pixel p to determine whether it is a corner. It examines a ring of 16 pixels, located on a Bresenham circle of radius 3, around p A pixel is said to be a corner if 9 contiguous pixels in the circle are either all brighter than, or all darker than p above/below the given threshold. The FAST algorithm is followed by a non-maximum suppression (NMS) of neighbouring pixels to avoid duplication of corners that are in proximity.

F. Orientation

The orientation of a keypoint is determined using an intensity centroid of the surronding pixels. First, the image is smoothened using a Gaussian filter of size 7x7. The centroid of image patch can be calculated by defining the moments in the x and y (m_{01} and m_{10} respectively) directions of the image patch. The moments give us the coordinates of the centroid C, which can be used to calculate the orientation θ of the keypoint.

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{1}$$

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

$$\theta = atan2(\frac{m_{01}}{m_{00}}, \frac{m_{10}}{m_{00}}) = atan2(m_{01}, m_{01})$$

G. BRIEF

BRIEF [22] outputs a description of each keypoint, as a unique identifier, for use in other frames (like a hash). The algorithm performs N comparisons between pairs of pixels and outputs an N-bit binary string as the output. The original ORB algorithm suggests choosing the pixel pairs as highly random with low correlation amongst the pairs, to maximize success and minimize false positives while matching keypoints.

The descriptor is calculated by selecting N pixel pairs (A,B) and setting the output bit if the intensity of pixel A is greater than pixel B, and reset otherwise.

$$\text{descriptor}[i] = I(A) > I(B)$$

where $i = 0...N$, A and B are sets of coordinates.

To achieve rotation invariance, the orientation angle of the keypoint, θ, is used to rotate the each pair in the BRIEF pattern using a rotation matrix.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2}$$

The pixels are then sampled using the new coordinates x' and y', to generate the descriptor. ORB uses a 256-bit descriptor, as it was determined to generate a unique enough descriptor while maintaining a low length. If a feature is detected in multiple frames, from different view points or orientations, their BRIEF patterns would be similar. Nearest neighbor search using Hamming distance is used to match the same feature across frames.

IV. Hardware Architecture

The architecture of the proposed accelerator is shown in Figure 3. ORB feature extraction and matching consumes 60-70% of the runtime of the algorithm, and, hence, it is offloaded onto hardware. The rest of the algorithm (Bundle adjustment) is run in software on the ARM cores. Based on the data from previous works, accelerating both the Bundle Adjustment and the ORB on the same FPGA would require a tremendous amount of resources. High Level Synthesis (HLS) has become popular, and has been used extensively in FPGA designs as they reduce the time required, and can sometimes be more efficient than RTL [23]–[26]. In this work, we use RTL for most of the design, and use HLS to define the communications between the CPU and FPGA.

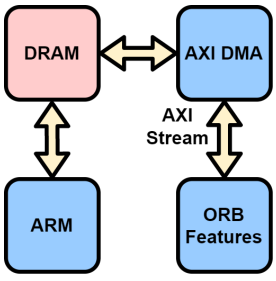

Fig. 3: Top level architecture of the ORB-SLAM implementation. Computation is partitioned between software and hardware.

The software and hardware are coupled using AXI-Stream via an AXI DMA. The feature extraction module operates as a streaming processor to take advantage of data re-use, without

requiring the use of DRAM to store intermediate state. The image cannot be stored in a frame buffer, as an entire image will not fit in most embedded FPGA's BRAM. The input image pixels are streamed from DRAM to the FPGA into a FIFO which are then dispatched to the ORB extractor. After the features are extracted, the descriptors are stored in a FIFO until transferred back to the CPU. The modules in hardware are fully pipelined, allowing it to process 1 pixel/cycle when the inputs are saturated. The feature extraction and matching can be completely independent of the back-end localization. This allows us to pipeline the dataflow between the front-end and back-end of the system. In the following sections, we will describe the hardware architecture of the ORB feature extraction module on the Programmable Logic.

A. Buffers

Throughout the pipeline we make use of buffers in two forms: line buffers, and window buffers. Pixels are streamed in a row major fashion; however, the kernels require input from multiple rows simultaneously to calculate the output for one pixel. We use line buffers as a mechanism to delay the input by N rows so that we have access to all N rows simultaneously. The linebuffers can be built from either BRAMs or the Shift Register primitives provided by LUTs. Window buffers are utilized to access pixel data in parallel. It is set up as 2-dimensional array of shift registers, and follows a linebuffer.

B. Image scalers

To generate our image pyramid, we downsample the input image at multiple levels. In order to avoid jagged edges and loss of information, the image has to be resampled. ORB-SLAM uses a scale factor of 1.2 for downsampling which can be expressed as a fraction $\frac{5}{6}$. Bilinear interpolation is used for resampled pixels that lie between the original pixels. The interpolation can be implemented using the 4 neighboring pixels with weights based on the location of the resampled pixel. The resampled pixel can be in one of 25 different locations, giving us 25 sets of weights. The weights can be calculated purely as a function of the original image's x and y coordinate modulo 6.

$$\begin{bmatrix} x_6 * y_6 & (5 - x_6) * y_6 \\ x_6 * (5 - y_6) & (5 - x_6) * (5 - y_6) \end{bmatrix}$$

where $x_6 = x$ modulo 6, $y_6 = y$ modulo 6. This makes an efficient down scaler only require 1 linebuffer and a 2x2 window buffer. If either x_6 or y_6 is equal to 5, the resampled pixel is not valid. Note that the downscaled pixels are only outputted every 5 out of 6 cycles, and no pixels are outputted on every 6th row.

The keypoints are written into the keypoint FIFO along with its (x,y) coordinate.

C. Orientation

The orientation requires a 37-row linebuffer as the input. It involves two parts. Calculating the moments m_{10} and m_{01} and using them to compute the arc tangent of $\frac{m_{10}}{m_{01}}$.

1) Moment: The naive approach to calculate the moment would involve calculating the individual parts in equation 1 every time. This would require storing a window buffer of 37x37 and either multiple cycles or multiple hardware resources in order to compute the moments. We implement a recursive approach to calculate the moments, where each moment can be computed from the last moment, the incoming column and the outgoing column.

2) Angle: Using the moments calculated from the previous block, we compute $\theta = arctan(m_{01}/m_{10})$ as the angle of the intensity centroid. The computation involves division and arc tangent, which are time-consuming operations in hardware. There exist CORDIC engines that take several cycles and multiple DSPs to compute arctangents with relatively precise outputs. However, CORDIC engines are restrictive in the size of the input x and y, which m_{01} and m_{10} generally exceed. Lookup Table (LUT) based methods can take 10s of cycles while using multiple BRAMs and DSPs to output an angle with a precision of less than 0.2 degrees. The feature extraction algorithm does not need to be precise, and the amount of resources required can be drastically reduced by discretizing the angles that a keypoint can have. We divide the circle into N_D sectors and round up the angle to the nearest sector. The sectors are defined by the line that divides the sectors in half.

We determine which quadrant the centroid belongs to based on the sign of m_{01} and m_{10}, and then shift the coordinates to the first quadrant by taking the absolute value of m_{01} and m_{10}. . To compute θ, we approximate $\theta = arctan(y/x) = arctan(m_{01}/m_{10})$ using LUTs and comparators to approximately satisfy the following equation.

$$abs(m_{10}) * tan(\theta) = abs(m_{01})$$

We compute $abs(m_{10}) * tan(\theta)$ by hardcoding the fixed point representations of all the sectors in the quadrant Four comparators in parallel check if $abs(m_{10}) * tan(\theta) > abs(m_{01})$ and use a priority encoder to determine the closest match for θ. Once determined, the module outputs the quadrant as well as θ to the BRIEF module. The architecture is illustrated in Figure 4.

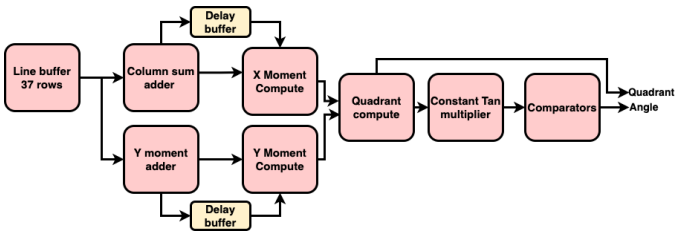

Fig. 4: Block diagram of Orientation module

D. rBRIEF

To compute BRIEF descriptors, a 37x37 window is constantly updated using the input from the linebuffer. The orientation module outputs the angle and the coordinates of the point it processed. The coordinates from the orientation

module is constantly compared with the head of the keypoint FIFO, and, if there is a match, the BRIEF module begins processing. While the keypoint is being processed, the 37x37 window is frozen until the descriptor is computed. After the descriptor is generated, the module needs 37 cycles to reload the window, before it is ready to process a new keypoint.

The BRIEF module consists of a rotator and a generator module. The rotator computes the rotated coordinates of the BRIEF pattern at 1 pattern/cycle using equation 2. The generator then takes 1 cycle to lookup the pixels from the window buffer and 1 more cycle to generate the descriptor. The rotator requires the values of $cos(\theta)$ and $sin(\theta)$ to be passed in by the dispatcher. As the angles are discretized, we can use small lookup tables to store the cosine and sine values for all the possible angles in a quadrant. The sin and cos values are stored in 8-bit fixed point. Based on the quadrant, the sign of the outputs is adjusted.

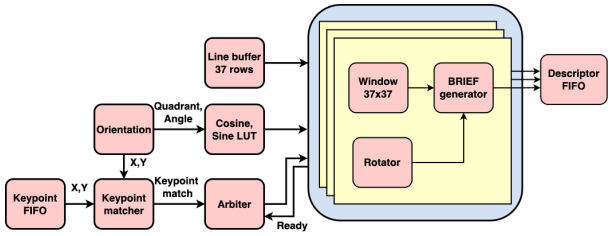

Fig. 5: Block diagram of BRIEF module

To eliminate the need for computing the rotated patterns, the original ORB paper proposes pre-computing the BRIEF patterns for all the possible angles and using a lookup-table during runtime to fetch the rotated coordinates. Each LUT would consume 256 pairs * 4 coordinates per pair * 5 bits per coordinate = 5120bits. If the angle is discretized into 32 sectors, the LUTs would requires 160Kbits of memory. The computation cost of 3 cycles and a few DSPs is insignificant compared to the resource usage of the LUT method, and hence we did not use the LUT method.

E. Comparison of various hardware architectures

Weberruss, et al. [8] and Tran, et al. [9] both use Harris Corner detector instead of FAST as the former is more accurate but computationally more expensive. Through experimentation we determined that for SLAM applications, high success rate of corner detection is not critical to the accuracy. The BRIEF modules we implemented are similar to that implemented in Weberuss, et al. [8]. The orientation computation is part of the BRIEF module, whereas we have the orientation module separated, avoiding duplication and saving on hardware resources. Similarly, we do not duplicate the cosine and sine LUTs, and instead pass the values in during dispatch. Fang, et al. [7] use frame buffers to store the image in 2 levels. We use a streaming method, so no frame buffers are required, saving a considerable amount of BRAM. Liu, et al. [18] utilize the same hardware for multiple scale levels. They have a dedicated image resizing module reading/writing directly from DRAM. This approach does not exploit the inherent parallelism available in processing multiple levels. It also incurs additional DRAM bandwidth for image resizing which we can avoid by using a streaming approach.

V. Experimental results and observations

Hardware setup: FSLAM is implemented on an Avnet Ultra96 development board. The system contains an Xilinx ZU3EG FPGA-SoC on-board with a quad-core ARM clocked at 1.5GHz, with the clock frequency of programmable logic running at 150MHz.

Dataset: We evaluate our system, FSLAM, using the TUM dataset [27] which provides RGB camera data, along with depth information, obtained from a handheld Kinect moving through various office environments. It is widely used in the visual SLAM community to evaluate the accuracy of algorithms. The datasets provide the ground truth of the pose of the camera obtained using a high-accuracy motion-capture system. The images are provided at a resolution of 640x480 at 30fps.

We use Absolute Trajectory Error (ATE) as the metric to evaluate accuracy. It measures the absolute difference between the ground truth poses and the estimated poses, and outputs the mean, median, and standard deviation of these differences. The root mean squared error (RMSE) of these differences is well suited to evaluating the accuracy of visual SLAM systems. The ORB-SLAM algorithm is not deterministic and has a variance of about 10% in the reported ATE across multiple runs using the same data. A lower ATE indicates better accuracy.

We evaluate the accuracy and the latency of feature extraction of our implementation on various datasets and compare it with eSLAM [18]. The latency of feature extraction in our implementation is around 2.5ms, which is 3.7x faster than eSLAM [18]. We also compare the resource utilization of our design as show in Table II, where we extracted the resource utilization of eSLAM from the paper [18].

TABLE II: FPGA resource utilization of ORB feature extraction

Implementation	LUTs	FFs	DSP	BRAM
eSLAM [18]	56594	67809	111	78
FSLAM	76424	101694	80	120

A. Determining sectors for orientiation approximation

The original ORB-SLAM implementation uses the OpenCV *fastatan2* function to compute the orientation. The function has a precision up to 0.3 degrees. In our hardware implementation, we discretized the angles into sectors as described in Section IV-C2. In order to determine the appropriate amount of sectors to discretize the angle space, we collect data on the effects of accuracy vs number of sectors. Figure 6 illustrates the accuracy comparison of discretizing the sectors when compared against the baseline *fastatan2*. Based on the data, the accuracy greatly reduces between 64 sectors to 32 sectors, and hence, we discretized our orientations to 64 sectors.

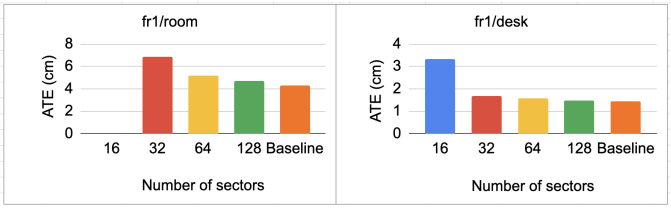

Fig. 6: Effect on accuracy when discretizing the angles to various sectors across 2 datasets fr1/desk and fr1/room. Baseline is CPU-based ORB SLAM. Missing data indicates that the algorithm could not converge.

B. *Effect of input pixel bit-depth on accuracy*

A large amount of Flip-Flop resources are consumed by the window buffers in the FPGA. A single ORB level uses approximately 38k FFs. In order to fit the design into the device, we explore reducing the bit-depth of each pixel at different stages of the ORB pipeline. We experimented with the pixels coming in to the FPGA from DRAM by truncating the lower bits after the FAST corner detection algorithm. We chose 2 datasets *fr1/room* and *fr1/desk* as they involve rapid movements, larger maps and loop closing to evaluate the impact on accuracy. The incoming pixels to the FPGA are 8-bit grayscale pixels.

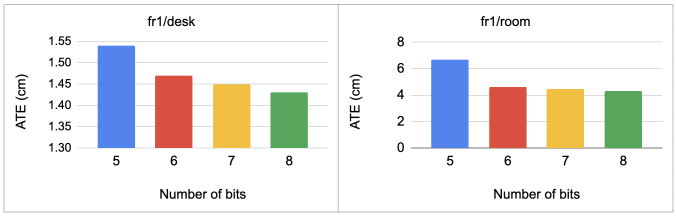

Fig. 7: Effect on accuracy with varying pixel bit-depth

Figure 7 illustrates the loss in accuracy as the bit-depth reduces. We chose to quantize our pixels to 6 bits per pixel which offers us a 25% reduction of FF resources while only dropping 3.9% in accuracy. However, this does not reduce the resource consumption of the BRAMs, as the way they are allocated is either as a 2Kx9bit array or a 4Kx4 bit array. The BRAM consumption is not a significant resource constraint, so this is not an issue.

C. *Accuracy evaluation*

We compare our work mainly with eSLAM [18] as it also implements an end-to-end ORB-based SLAM system on an SoC FPGA. We also compare our work against the original ORB-SLAM which was designed to run on a CPU. The runtime of our tracking thread on software is 30ms, while eSLAM's is around 20ms. Based on an initial analysis, our tracking thread is far more complex than that of eSLAM. The exact algorithm is not specified in the paper, but it appears that they only perform motion-only BA without using local BA or loop closing. To further investigate the cause for the discrepancy in accuracy between eSLAM and CPU-based ORB-SLAM, we run ORB-SLAM with local BA and loop closing disabled on the various datasets. Figure 8 plots the normalized accuracy for each dataset against the original ORB-SLAM, while Table III contains the raw data. We observe that without local BA, there is a large drop in accuracy in the datasets *fr2/desk* and *fr3/office*. Both of these involve extensive translatory motion and generate a large enough map such that keyframes are spread out enough where local BA makes a difference.

Loop closure also has an impact on those datasets where loops exist. In our case, *fr1/room* and *fr3/office*, both contain large loops. As seen in the chart, these two datasets have a significant accuracy drop, while the other datasets remain the same. Looking at the accuracy of *fr1/room* in particular, the accuracy significantly drops when removing local bundle adjustment and loop closing. This dataset has the camera moving across a large open room, eventually ending back at the starting point.

eSLAM also proposed a novel rotationally-symmetrical BRIEF descriptor (RS-BRIEF) which reduces the hardware complexity in rotating the BRIEF descriptors by changing it from a trignometric operation to a bit rotation operation. This approach saves computation time and resources; however, it also leads to great loss in accuracy. The BRIEF patterns chosen in the ORB-SLAM algorithm were deliberate to maximize the accuracy as described in Section III-G. We do not implement RS-BRIEF in our work as we would have to retrain the vocabulary used to describe the BRIEF descriptor. The amount of resources saved by using RS-BRIEF does not warrant the great drop in accuracy.

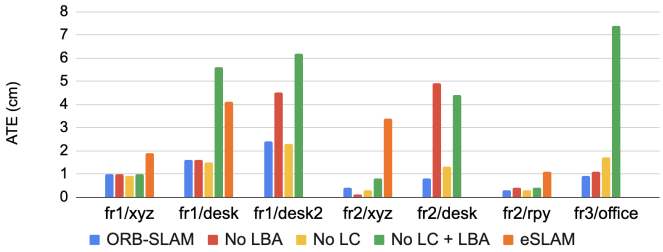

Fig. 8: Tracking accuracy of datasets with Local Bundle Adjustment (LBA) and Loop Closing (LC)

TABLE III: ATE (in cm) of various modifications to ORB-SLAM Tracking. (X indicates data was not available)

Dataset	ORB-SLAM	No LBA	No LC	No LC or LBA	eSLAM
fr1/xyz	1.0	1.0	0.9	1.0	1.9
fr1/desk	1.6	1.6	1.5	5.6	4.1
fr1/room	4.3	7.3	10.6	20.2	11.1
fr1/desk2	2.4	4.5	2.3	6.2	X
fr2/xyz	0.4	0.1	0.3	0.8	3.4
fr2/desk	0.8	4.9	1.3	4.4	X
fr2/rpy	0.3	0.4	0.3	0.4	1.1
fr3/office	0.9	1.08	1.7	7.4	X

Figure 9 illustrates the accuracy comparison of our work against eSLAM and the original CPU implementation of ORB-SLAM. Our work modified ORB-SLAM with the hardware optimizations described in Sections V-A and V-B. On average, our work is around 5-10% less accurate compared to the CPU version of ORB-SLAM and 50-70% more accurate when compared to eSLAM. This improvement in accuracy compared to eSLAM is due to the addition of Local BA and Loop closing threads which greatly improves accuracy in complex datasets. The degradation in accuracy compared to the original ORB-SLAM is acceptable as it is a relative comparison to the error rate. The absolute difference in observational error (averages around 0.01cm-1cm based on the dataset) is insignificant when compared to the total distance travelled (1.5m-20m).

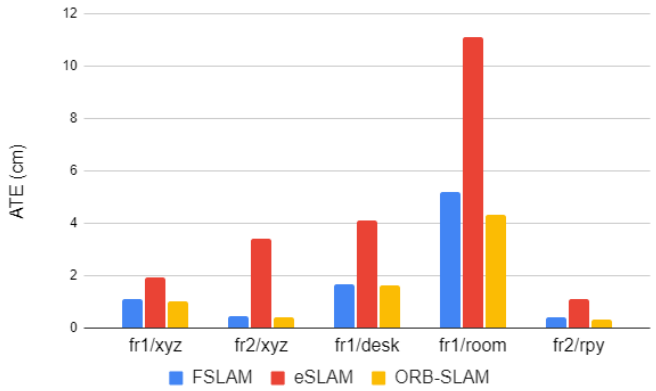

Fig. 9: Accuracy comparison amongst different platforms on various datasets

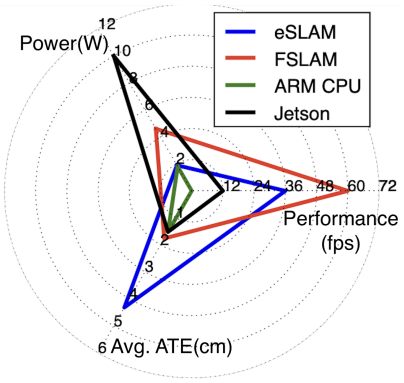

Fig. 10: Comparison of metrics on various platforms

D. Performance and Power Evaluation

The performance of FSLAM is compared with software implementations on the integrated ARM processor of the SoC, a desktop with an Intel i5 processor coupled with a NVIDIA 2070S, and a Jetson Xavier evaluation platform that runs an embedded SoC. The Jetson SoC contains an 8-core ARM CPU and a 512-core GPU. The ARM CPU mentioned in the table runs the entire algorithm on the SoC's CPU. We also compare our work against the state-of-the-art eSLAM [18]. Table IV shows the runtime comparison of our work against the different evaluation platforms. The performance is a bit worse than the desktop GPU + CPU variant, however our system consumes orders of magnitude less power allowing it to be used in mobile environments. We are able to achieve a 1.55x speedup in comparison to the desktop CPU, and a 1.35x speedup compared to eSLAM. The table also shows the power consumed by the various platforms. Our work is 43x, 9.7x more power efficient when compared to desktop systems with and without a GPU respectively. Compared to the embedded platforms, our work is 2.1x more power efficient than the Jetson, while consuming 2.5x more power than the ARM CPU standalone. We consume 2.4x more power than eSLAM as the higher throughput and accuracy are reflected in the resource consumption, and computational complexity. Figure 10 illustrates the trade-offs of power, performance and ATE across various platforms.

TABLE IV: Framerate, power consumption and accuracy of ORB-SLAM on various platforms

Platform	Framerate	Speedup	Power	Avg. ATE
Desktop CPU	40	1x	45W	1.56cm
Desktop GPU	70	1.75x	200W	1.54cm
Jetson	22	0.55x	10W	1.54cm
ARM CPU	7	0.175x	1.8W	1.54cm
eSLAM [18]	45	1.125x	1.9W	4.32cm
FSLAM	62	1.55x	4.6W	1.76cm

VI. Conclusion

In this paper, we presented FSLAM, an end-to-end implementation of ORB-SLAM on an FPGA to ensure both real-time performance and high accuracy. In order to meet performance requirements, the ORB accelerator is formatted as a streaming processor, which avoids the utilization of memories, enables data re-use and processes 1 pixel per cycle. To facilitate fitting the high-performance accelerator into an embedded device, data-driven hardware optimizations were made by trading-off hardware resources for slightly lower accuracy. The optimizations were made based on experimental observations to maximize resource savings, while minimizing accuracy loss. Leveraging the above factors, the evaluation results show that we achieve a 1.55x speedup compared to a desktop CPU while maintaining a reasonable accuracy, and 1.35x speedup compared to previous works [18] while averaging a 2x improvement in accuracy across multiple datasets. We also show a 10x, and 40x improvement in power over desktop systems without and with a GPU respectively. Our work is open sourced [28].

Acknowledgements

This work is supported in part by the AMD/Xilinx Center of Excellence and the HACC (Heterogenous Accelerated Compute Cluster) at University of Illinois at Urbana-Champaign.

REFERENCES

[1] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[2] 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.

[3] C. Campos, R. Elvira, J. J. Gomez, J. M. M. Montiel, and J. D. Tardos, "ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[4] S. Aldegheri, N. Bombieri, D. D. Bloisi, and A. Farinelli, "Data flow orb-slam for real-time performance on embedded gpu boards," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 5370–5375.

[5] X. Zhang, C. Hao, H. Lu, J. Li, Y. Li, Y. Fan, K. Rupnow, J. Xiong, T. Huang, H. Shi *et al.*, "Skynet: A champion model for dac-sdc on low power object detection," *arXiv preprint arXiv:1906.10327*, 2019.

[6] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen, "Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[7] W. Fang, Y. Zhang, B. Yu, and S. Liu, "Fpga-based orb feature extraction for real-time visual slam," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 275–278.

[8] J. Weberruss, L. Kleeman, D. Boland, and T. Drummond, "Fpga acceleration of multilevel orb feature extraction for computer vision," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[9] P. Tran, T. H. Pham, S. K. Lam, M. Wu, and B. A. Jasani, "Stream-based orb feature extractor with dynamic power optimization," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 94–101.

[10] S. Qin, Q. Liu, B. Yu, and S. Liu, "π-ba: Bundle adjustment acceleration on embedded fpgas with co-observation optimization," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 100–108.

[11] Q. Liu, S. Qin, B. Yu, J. Tang, and S. Liu, "π-ba: Bundle adjustment hardware accelerator based on distribution of 3d-point observations," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1083–1095, 2020.

[12] Y. Wu, L. Luo, S. Yin, M. Yu, F. Qiao, H. Huang, X. Shi, Q. Wei, and X. Liu, "An fpga based energy efficient ds-slam accelerator for mobile robots in dynamic environment," *Applied Sciences*, vol. 11, no. 4, p. 1828, 2021.

[13] C. Yu, Z. Liu, X.-J. Liu, F. Xie, Y. Yang, Q. Wei, and Q. Fei, "Ds-slam: A semantic visual slam towards dynamic environments," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1168–1174.

[14] W. Liu, B. Yu, Y. Gan, Q. Liu, J. Tang, S. Liu, and Y. Zhu, "Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 479–493.

[15] K. Boikos and C.-S. Bouganis, "A high-performance system-on-chip architecture for direct tracking for slam," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[16] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *European conference on computer vision*. Springer, 2014, pp. 834–849.

[17] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, "Pisces: power-aware implementation of slam by customizing efficient sparse algebra," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[18] R. Liu, J. Yang, Y. Chen, and W. Zhao, "eslam: An energy-efficient accelerator for real-time orb-slam on fpga platform," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 193.

[19] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[20] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—a modern synthesis," in *International workshop on vision algorithms*. Springer, 1999, pp. 298–372.

[21] E. Rublee, V. Rabaud, K. Konolige, and G. R. Bradski, "Orb: An efficient alternative to sift or surf." in *ICCV*, vol. 11, no. 1. Citeseer, 2011, p. 2.

[22] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*. Springer, 2010, pp. 778–792.

[23] D. Chen, J. Cong, and J. Xu, "Optimal module and voltage assignment for low-power," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 850–855.

[24] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong, "Multilevel granularity parallelism synthesis on fpgas," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 178–185.

[25] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 741–755.

[26] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 54–64.

[27] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[28] V. Vemulapati, "Fslam," *Github Repository*, 2022. [Online]. Available: https://github.com/vvemulapati/FSLAM