Reconfigurable System-on-Chip Architectures for Robust Visual SLAM on Humanoid Robots

MARIA RAFAELA GKEKA, University of Thessaly, Greece
ALEXANDROS PATRAS, University of Thessaly, Greece
NIKOLAOS TAVOULARIS, Foundation for Research and Technology - Hellas (FORTH), Greece
STYLIANOS PIPERAKIS, Foundation for Research and Technology - Hellas (FORTH), Greece
EMMANOUIL HOURDAKIS, Foundation for Research and Technology - Hellas (FORTH), Greece
PANOS TRAHANIAS, Foundation for Research and Technology - Hellas (FORTH), Greece
CHRISTOS D. ANTONOPOULOS, University of Thessaly, Greece
SPYROS LALIS, University of Thessaly, Greece
NIKOLAOS BELLAS, University of Thessaly, Greece

Visual Simultaneous Localization and Mapping (vSLAM) is the method of employing an optical sensor to map the robot's observable surroundings while also identifying the robot's pose in relation to that map. The accuracy and speed of vSLAM calculations can have a very significant impact on the performance and effectiveness of subsequent tasks that need to be executed by the robot, making it a key building component for current robotic designs. The application of vSLAM in the area of humanoid robotics is particularly difficult due to the robot's unsteady locomotion. This paper introduces a pose graph optimization module based on RGB (ORB) features, as an extension of the KinectFusion pipeline (a well-known vSLAM algorithm), to assist in recovering the robot's stance during unstable gait patterns when the KinectFusion tracking system fails. We develop and test a wide range of embedded MPSoC FPGA designs, and we investigate numerous architectural improvements, both precise and approximation, to study their impact on performance and accuracy. Extensive design space exploration reveals that properly designed approximations, which exploit domain knowledge and efficient management of CPU and FPGA fabric resources, enable real-time vSLAM at more than 30 fps in humanoid robots with high energy-efficiency and without compromising robot tracking and map construction. This is the first FPGA design to achieve robust, real-time dense SLAM operation targeting specifically humanoid robots. An open source release of our implementations and data can be found in [1].

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**; • **Theory of computation → Approximation algorithms analysis**; • **Computer systems organization →** *Robotics.*

Authors' addresses: Maria Rafaela Gkeka, margkeka@uth.gr, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece; Alexandros Patras, patras@uth.gr, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece; Nikolaos Tavoularis, tavu@ics.forth.gr, Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece; Stylianos Piperakis, spiperakis@ics.forth.gr, Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece; Emmanouil Hourdakis, ehourdak@ics.forth.gr, Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece; Panos Trahanias, trahania@ics.forth.gr, Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece; Christos D. Antonopoulos, cda@uth.gr, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece; Spyros Lalis, lalis@uth.gr, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece; Nikolaos Bellas, nbellas@uth.gr, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece.

1539-9087/2022/11-ART $15.00
https://doi.org/10.1145/3570210

Additional Key Words and Phrases: Simultaneous Localization and Mapping, Humanoid Robots, FPGA, Approximate Computing

1 INTRODUCTION

Humanoid robots are expected to be used in a close relationship with humans in daily life, and to serve the needs of elderly and physically handicapped people [12]. These robots must be able to collaborate in a dynamic and unstructured environment performing a wide variety of tasks. The proliferation of such autonomous robots has created the need to construct highly accurate maps of their observed environment and to track their position and their trajectory within these maps. This is a prerequisite for robot tasks such as path planning and obstacle avoidance.

The availability of multisensory information such as RGB-D cameras, laser scanners (lidars) and Inertial Measurement Units (IMUs) used to imitate human proprioception, is critical for the robots ability to build a consistent map and to determine its location within that map, when placed at an unknown position in an unknown environment. The construction of the environmental model (mapping) while simultaneously positioning the robot within that environment (localization) is collectively called *Simultaneous Localization and Mapping (SLAM)* [14, 17, 19, 30, 32, 41, 42, 53, 55, 68]. vSLAM algorithms use cameras as sensors. Dense vSLAM algorithms use all pixels of the input frame for map reconstruction and provide the potential for richer 3D scene modeling and interaction with the environment. Besides robot navigation [3, 25, 29, 64], SLAM algorithms are widely used in Augmented/Virtual Reality [34, 52], unmanned aerial vehicles (UAVs) [16, 22, 54], and autonomous driving [11].

The problem of mapping and localizing a walking humanoid is hard to solve using existing vSLAM algorithm and remains to-date an open-research problem. Difficulties arise from the high number of Degrees of Freedom (DoF), the highly nonlinear dynamics due to the vertical alignment of the kinematic chain, the under-actuation occurring during the walking cycle, and the unilateral Ground Reaction Forces (GRFs) and Ground Reaction Torques (GRTs) the robot experiences while walking. These effects commonly give rise to leg slippage, sudden acceleration caused by leg impacts, and unmodeled/undesired compliance, which cannot be effectively captured with purely kinematic-inertial estimation methods [46]. Using a conventional SLAM algorithm to estimate the pose (position and orientation) of humanoid robots would, thus, result into a large number of untracked frames and large errors of the estimated trajectory.

To address these problems and optimize the quality of pose estimation, we propose PG-SLAM, a novel algorithmic extension to KinectFusion (a baseline dense vSLAM algorithm) [30], in which we optimize the pose estimation of the robot through a pose graph (PG) correction algorithm. The initial pose estimation of KinectFusion is introduced to the PG correction pipeline, which produces a more accurate estimate through a pose graph optimization phase using RGB features. This technique enhances the robustness of SLAM and reduces the number of untracked frames, even to zero for some trajectories. However, such algorithmic extensions to the baseline KinectFusion pipeline further exacerbate the already high computational and energy costs of vSLAM, since, to reduce the accumulated error, the pose graph pipeline periodically inserts expensive corrective computations to recover from pose graph errors.

In an effort to tackle this challenge and enable real time performance of the new algorithm, our work proposes new MPSoC FPGA-based designs for humanoid SLAM computing that combine application-specific optimizations with approximate computing. Approximate computing has been shown to accelerate computations and reduce energy requirements in various application domains [38]. Our objective is to achieve real time SLAM performance without exceeding the number of untracked frames by more than a threshold, potentially, derailing the robot off its trajectory. We show that a combination of precise and approximate optimizations and tuning of algorithmic parameters provide a speedup of up to 15.7X compared with the precise FPGA implementation without violating the tight constraints on the number of untracked frames.

The main contributions of our work are the following:

- We introduce PG-SLAM, a novel algorithmic extension to KinectFusion, which is based on periodic correction of the pose graph of a humanoid robot to reduce the accumulated pose error and the number of untracked frames.
- To implement PG-SLAM, we consider a plethora of precise and approximate optimizations, and, based on those, we introduce new parameterizable MPSoC FPGA-based implementations spanning the performance vs. accuracy space.
- We evaluate these implementations to show that the proposed methodology can optimize performance and achieve real time operation while limiting accuracy loss within reasonable bounds.
- We provide a mechanism to rank the impact of each optimization, and we show how this mechanism can be used to validate our MPSoC design decisions.
- Finally, we evaluate the effects of depth holes of the RGB-D cameras on the accuracy of the PG-SLAM algorithm, and we propose techniques to mitigate their effects.

The remainder of the paper is organized as follows: In Section 2 we overview previous work on SLAM computing and SLAM implementations. Section 3 describes the algorithmic details of the PG-SLAM algorithm, whereas section 4 describes the parameterized architectures and the approximations for each kernel and for the whole design. Section 5 presents the experimental evaluation and Section 6 concludes the paper.

2 RELATED WORK

The following paragraphs provide an in-depth discussion of the state of the art on (i) humanoid localization and mapping, and (ii) implementations of SLAM algorithms with emphasis on FPGAs. Table 1 presents a synopsis of such implementations on various computing platforms. Note that there is no prior work exploring SLAM algorithms for humanoids on FPGA platforms.

2.1 Algorithms for Humanoid SLAM

In [46], the authors propose an odometric localization method that estimates the torso position of a humanoid robot, by integrating IMUs and monocular vision. In [60] the authors propose a feature tracking approach based on stereo vision, while [47] proposes a vSLAM system for footstep planning by tracking 3D flow image features using stereo depth maps and RANSAC.

Apart from localization, the provision of a high-resolution map is often necessary to support the robot's locomotion and navigation modules. For example in [71] the authors integrate ElasticFusion with pose tracking on a humanoid, to create an RGB-D SLAM system that can map its surroundings despite the presence of humans in the scene. [50] demonstrates how the use of kinematic-inertial state estimation can be employed to improve the mapping process of KinectFusion [30]. PG-SLAM uses RGB-D input to build a dense map (similar to KinectFusion and ElasticFusion) and simultaneously constructs a pose graph based on ORB features to dynamically correct accumulated pose errors.

2.2 Dense SLAM Accelerators

Recently, a few FPGA implementations have focused on dense SLAM. Dense SLAM algorithms use all the pixels of the input images to achieve an information-richer 3D scene reconstruction. Authors in [26] evaluate a large space of precise and approximate MPSoC FPGA architectures of KinectFusion [30] on a Xilinx UltraScale+ ZCU102 board and show that approximations are necessary to achieve real-time performance (at almost 30 fps). They also propose a systematic methodology to rank the impact of each optimization on the performance and output error of KinectFusion, and use it as an optimization selection mechanism. Similarly, [23] performs extensive design space exploration for the InfiniTAM algorithm [31] (which is derived from KinectFusion) by developing a large number of parameterized architectures for each of the dense SLAM components. In that work, authors only focus

on performance, without exploring any accuracy vs. performance trade-offs. The same authors explore both the GPU and the Altera Stratix V FPGA to implement the tracking and integration components and achieve 28 fps real time performance [24].

2.3 Sparse SLAM Accelerators

Instead of using all pixels of the frame, sparse SLAM algorithms maintain only a small selection of features to continuously track a sparse map of the surrounding environments. The reduction in computational requirements and power dissipation make these algorithms appealing for implementation in embedded, power-constrained environments. Sparse SLAM algorithms are typically used only for agent localization.

ORB-SLAM is a feature-based sparse SLAM that operates in real time without the need for acceleration in indoor and outdoor environments [41]. The hardware implementation of the ORB-SLAM feature extractor (which accounts for 50% of total ORB-SLAM computation) presented by Fang et al. outperformed ARM Krait by 51% and Intel Core i5 by 41% in computation latency [20]. [36] proposes eSLAM, an energy-efficient FPGA implementation of feature extraction and feature matching stages. Compared with Intel i7 and ARM Cortex-A9 CPUs, eSLAM achieves up to 3X and 31X frame rate improvement, as well as up to 71X and 25X energy efficiency improvement, respectively. Finally, Abouzahir et al. perform a complete study of the processing time of four SLAM algorithms on popular embedded devices. They demonstrate that Fast-SLAM2.0 is an appealing algorithm for FPGA implementation due to its massive parallelization potential [2].

EKF-SLAM is a class of algorithms which utilizes the extended Kalman filter (EKF) for SLAM. Typically, EKF-SLAM algorithms are feature based, and use the maximum likelihood algorithm for data association. Bonato et al. present the first FPGA-based architecture for the EKF algorithm that is capable of processing two-dimensional maps containing up to 1.8 k features at 14 fps, a three-fold improvement over a Pentium M 1.6 GHz, and a 13-fold improvement over an ARM920T 200 MHz [10]. A somewhat more recent work by Vincke et al. implements the EKF-SLAM algorithm on a multi-core ARM processor with a SIMD coprocessor and a DSP core [67]. Tertei et al. improve the EKF-SLAM performance on the Virtex5 [62] and Zynq 7020 [63] FPGAs achieving over 30Hz.

A low computational complexity Visual-Odometry SLAM (VO-SLAM) algorithm is proposed and implemented by Gu et al. [27]. They adopt an embedded system design comprising a master processor and multiple matrix accelerators implemented on a DE3 board (Altera Stratix III). Onboard tests suggest that the system can process a global map of 30000 feature points at 31 fps, and achieves 10x energy saving in processing each frame compared with Intel i7. [44] augments vSLAM with MEMS Inertial Measurement Units (IMUs) to provide measurements of angular velocity and linear acceleration. The system implements the Harris and FAST corner algorithms for feature detection on the fabric of Zynq-7020 MPSoc FPGA. Finally, [21] designs an FPGA-based SLAM architecture to achieve real-time 3D mapping using LiDAR input sensors.

2.4 Semi-dense SLAM Accelerators

Semi-dense SLAM provides a more dense and information-rich representation compared to sparse methods. Large-Scale Direct Monocular SLAM (LSD-SLAM) is a widely-used semi-dense SLAM algorithm. In [9], LSD-SLAM is accelerated on an FPGA SoC, achieving more than 60 fps on a 640x480 input frame. Previous work by the same authors describes an FPGA accelerator only for semi-dense tracking on embedded platforms [8].

The CNN-based feature-point extraction methods such DeepDesc [57] and SuperPoint [18] have made significant strides of progress in both feature-point detection and descriptor generation compared with earlier methods. Xu et al. propose an FPGA design to accelerate an optimized version of SuperPoint CNN-based feature extraction method achieving real-time (20 fps) execution on the ZCU102 platform [70].

2.5 Approximate SLAM

Approximate computing has been used to accelerate SLAM implementations. In SLAMBooster, the degree of approximation is dynamically adjusted during the motion of the robot [48]. For example, the accuracy of the SLAM algorithm is increased when the surface is detected to be smooth (e.g. when the scene represents a flat field) or in case a sudden pose change is detected between successive frames. In a follow up work, the same authors generalized their approximation methodology in four SLAM algorithms and evaluated them across different embedded platforms [49]. The improvement in computation time and energy usage per frame for KinectFusion is 77% and 40% on ODROID XU4, and 57% and 36% on Jetson TX2 boards with acceptable quality loss in localization and mapping accuracy over a variety of inputs. [45] studies the performance impact of reduced-precision floating-point arithmetic on SLAM algorithms. A very low-power ASIC design for real-time visual inertial odometry (VIO) targeting nano-drones has been announced [59]. The chip, fabricated at 65nm CMOS technology, uses inertial measurements and mono/stereo images to estimate the drone's trajectory and a 3D map of the environment. It can process 752×480 stereo images at 20 fps consuming an average power of merely 2 mW.

2.6 Contribution

We extend the previous works in several directions. Firstly, we provide algorithmic extensions to the baseline KinectFusion algorithm to dynamically and periodically correct the pose estimation of a humanoid. Then, we parameterize the PG-SLAM algorithm in terms of precise and approximate parameterization knobs, and we adjust the values of these knobs to implement various versions of the PG-SLAM pipeline on a MPSoC FPGA spanning the performance vs. accuracy space area. To our knowledge, this work is the first study and FPGA implementation of a SLAM algorithm customized specifically for humanoid robots.

3 ROBUST VISUAL SLAM ALGORITHM

In this section we describe the PG-SLAM pipeline that integrates KinectFusion with pose graph optimizations to recover from tracking failures. We provide details about its implementation and provide an introductory performance analysis when PG-SLAM is executed in a multicore CPU. Figure 1 shows the system architecture.

3.1 Acquisition

The pipeline system acquires two images from an RGB-D camera, a 16-bit image for depth, and an 8-bit, 3-channel image for color. The data are synchronized and packed on a single RGB-D frame as $I : \Omega \rightarrow \mathbb{R}^4$ where $\Omega \subset \mathbb{R}^2$ is the 2D image plane. The depth frame is passed to KinectFusion, which estimates a relative pose for two consecutive frames, and maintains a representation of a dense map. The RGB frame is used for feature detection and pose graph optimization.

3.2 KinectFusion

KinectFusion uses depth frames to perform real-time localization and dense mapping [30]. It is a closed-loop algorithm which continuously embeds new depth information to a partially constructed 3D view of a globally consistent map. As background information, this subsection describes the main components of KinectFusion (blue blocks in Figure 1).

Bilateral filter is a stencil-based, edge-preserving filter that reduces the effects of noise and invalid depth values in the input depth image [66]. It uses a 5x5 array for convolution, whose coefficients are a combination of spatial distance and image intensity difference from the central pixel p. Pixels within the 5x5 window whose value are very dissimilar to the value of p do not contribute much to the output value of p.

Tracking estimates the 3D pose of the agent by aligning the source depth image with the 2D projection of the currently reconstructed reference model (i.e. the 2D voxel map produced by raycasting). It uses a three-level

Table 1. Previous work on SLAM implementations.

SLAM type	Ref.	SLAM Algorithm	Input Sensors	Performance & Platform	Comments
Dense SLAM	This work	PG-SLAM	RGB-D (320x240)	1.59 fps (ARM A53) 37.8 fps (ZCU102 SoC)	Only FPGA SLAM impl. for humanoids
	[26]	KFusion [30]	Depth (320x240)	2.38 fps (ARM A53) 27.5 fps(ZCU102 SoC)	Extensive use of approx. accelerators
	[23]	Infinitam [31]	Depth (320x240)	0.65 fps (ARM A9) 1.22 fps (Teras. DE1) 44 fps (Terasic DE5)	Extensive use of approx. accelerators
	[24]	KFusion [30]	Depth (320x240)	26-28 fps(Terasic DE5)	All kernels in GPU. ICP in FPGA.
	[43]	KFusion [30]	Depth (320x240)	135 fps (TITAN) 96 fps (GTX-870M) 5.5 fps (ODROID-XU3)	CUDA implementation
	[49]	KFusion [30]	Depth (320x240)	13.7 fps (ODROID-XU4) 38.4 fps (Jetson X2)	Online (dynamic) approximations
Sparse SLAM	[36]	ORB-SLAM [41]	RGB-D (320x240)	1.8 fps (ARM A9) 55.87 fps (Zynq XCZ7045)	Feature Extraction & Matching
	[20]	ORB-SLAM [41]	RGB-D (320x240)	33 fps (ARM Krait) 67 fps(Stratix V)	Frame rate refers to Feature Extraction
	[2]	FastSLAM 2.0 [39]	Laser Scanner	5.8 fps (ODROID-XU4) 102.14 fps (Arria 10 SoC)	
	[27]	VO-SLAM [27]	Stereo Cameras	41 fps (i7-3770K) 31 fps (DE3 with Stratix III)	
	[59]	VI-odometry	Stereo & IMU (752x480)	20-171 fps (ASIC)	2mW power (min) Used in Nanodrones
	[10]	EKF-SLAM [58]	Stereo Cameras	4.5 fps (Pentium M), 1.1 fps (ARM920T) 14 Hz (Celoxica RC250)	
	[62]	EKF-SLAM [58]	Monocular & IMU	44.39 Hz (Virtex5 XC5VFX70T)	
	[63]	EKF-SLAM [58]	Monocular & IMU	>30 Hz (Zynq 7020)	
	[21]	LiDAR-based with TSDF	LiDAR & IMU	3.58 Hz (i7-6770HQ) 26.3 Hz (UltraScale+ XCZU15EG)	
	[49]	ICE-BA [35]	Depth (320x240)	17.25 fps (ODROID-XU4) 25.64 fps (Jetson X2)	Online dynamic approximations
Semi-dense SLAM	[7]	LSD-SLAM [19]	Depth (320x240)	2.27 fps (ARM A9) 4.55 fps (Zynq 7020)	
	[8]	LSD-SLAM [19]	Depth (320x240)	22fps (Zynq 7020)	Semi-dense track only
	[9]	LSD-SLAM [19]	Depth (640x480)	60 fps (i7-4770) 60 fps (Zynq ZC706)	
	[70]	CNN-based Feat.Extract.	Visual Sensor	20 fps (Zynq ZC102)	Frame rate refers to Feature Extraction

image pyramid by sub-sampling the filtered depth image. At the front-end of tracking (not shown in Figure 1), the **depth2vertex** function transforms each pixel of the pyramid images into a 3D point (vertex), thus forming a point cloud, and the **vertex2normal** function computes the normal vectors for each vertex of the point cloud.

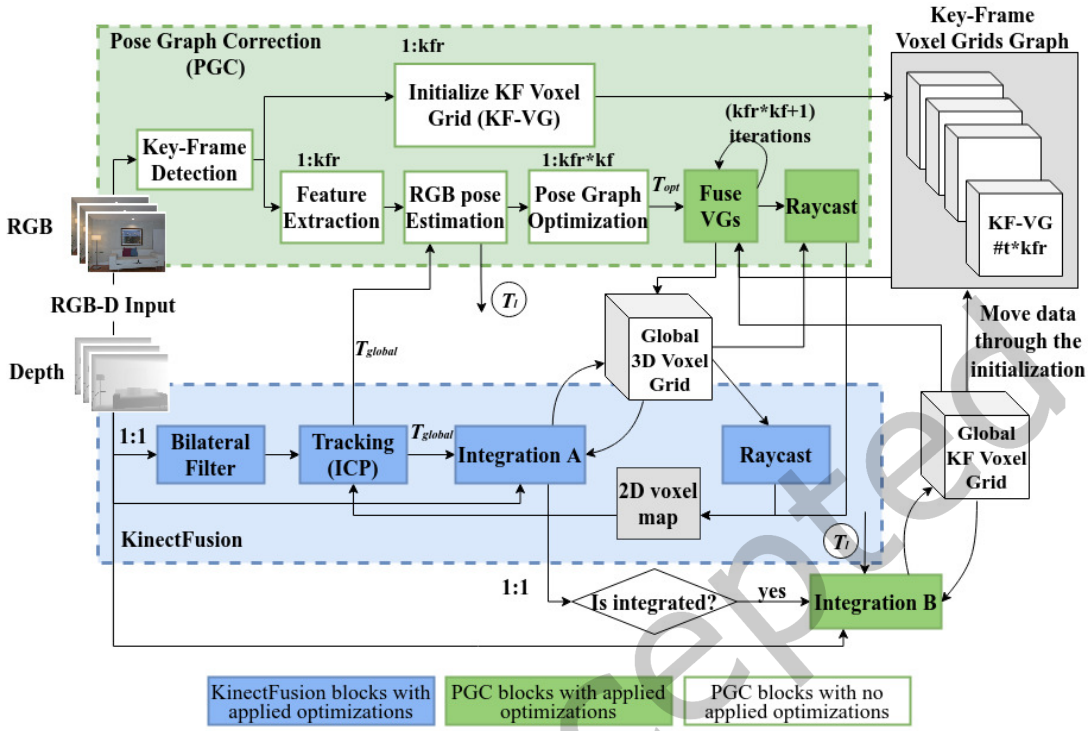


Fig. 1. The proposed vSLAM pipeline. Blue blocks indicate the original KinectFusion pipeline, whereas green blocks indicate the additional pose graph pipeline proposed in this paper. The notation 1:N means that a component is executed once every N frames. The FuseVG component is invoked $kf * kfr + 1$ times every $kfr * kf$ frames. After processing each input frame, the algorithm outputs the estimated pose T_{global} and the reconstructed map of the environment (Global 3D Voxel Grid).

Tracking is based on the Iterative Closest Point (ICP) algorithm which tries to pair the pixels of the source frame to the pixels of the reference frame [5]. ICP iteratively applies a combination of translation and rotation transformations needed to maximize the overlap between the source and the reference point cloud. Starting from low-resolution images in the pyramid, ICP estimates the overlap between the two point clouds and updates the pose when the overlap does not exceed a user-defined threshold (icp). When a new pose occurs or the ICP iteration limit is reached, the frame is tagged as untracked and the estimated pose is rejected, if the number of pixels that cannot be paired is more than the 85% (this is a constant threshold) of the total number of the pixels of frame.

Integration fuses the output of the tracker into the current 3D reconstructed model. It utilizes a 3D voxel grid as the data structure to represent the global map, employing a truncated signed distance function (TSDF) to represent 3D surfaces [15]. TSDF values are positive in front of the surface and negative behind the surface of an object so that the surface boundary is defined by the zero-crossing where the values change sign (Figure 2a). The updates of the $TSDF_i(p)$ voxel values and the weights $W_i(p)$ at position $p(x, y, z)$ of frame i are computed with

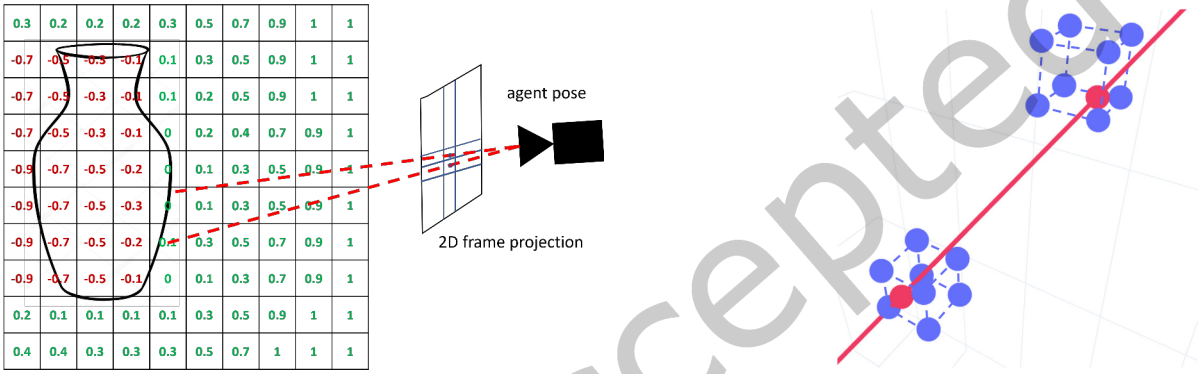
the following equations:

$$SDF_i(p) = \min(1, diff_i(p)/mu) \tag{1}$$

$$TSDF_i(p) = \begin{cases} \frac{W_{i-1}(p)*TSDF_{i-1}(p)+SDF_i(p)}{W_{i-1}(p)+1}, & \text{if } diff_i(p) > -mu. \\ TSDF_{i-1}(p), & \text{otherwise.} \end{cases} \tag{2}$$

$$W_i(p) = \min(W_{max}, W_{i-1}(p) + 1) \tag{3}$$

where $TSDF_{i-1}(p)$ and $W_{i-1}(p)$ are the TSDF and weight values of the voxel p at the frame $i-1$, mu is the maximum truncation distance, W_{max} is a constant parameter equal to 100, and $diff_i(p)$ describes the difference between the actual depth measurements and a metric distance from the estimated camera position to p.



(a) Simplified view of the voxel grid in 2D space

(b) Trilinear interpolation

Fig. 2. Raycast is based on traversing imaginary rays towards the voxel grid and interpolating fractional TSDF points.

Raycast is a basic computer graphics algorithm used to render three-dimensional scenes to two-dimensional images. Specifically, virtual light rays are cast from the focal point of the agent's camera through each one of the camera pixels towards the 3D voxel grid at an angle that is determined by the pose of the agent (Figure 2a). These rays are traced step-wise from the focal point until they intercept an object in the voxel grid or until they exit the voxel grid which denotes the absence of an object in that path. Raycast employs trilinear interpolation to compute the missing fractional TSDF value at each step along the ray using the 8 adjacent TSDF values (Figure 2b). According to the definition of the TSDF, a small negative interpolated value indicates that the ray has just intercepted the object moving to a spot that is not visible from the camera. In that case, raycast backtracks (using a smaller step) along the ray trying to reach the surface of the object, i.e. an interpolated value which is very close to zero. Raycast updates the vertex and normal maps that identify the implicit surface to the current estimate of the camera position.

3.3 Pose Graph Correction

The proposed pose graph correction pipeline uses the RGB image stream to introduce additional constraints at selected keyframes, and rectify the pose estimates of KinectFusion. It formulates the observation error of 2D features as constraints in a factor graph, and optimizes the graph to compute a corrected pose for every keyframe. For each new keyframe pose, we initialize an intermediate keyframe voxel grid (called KF-VGs in Figure 1) which is used to apply periodic corrections (every $kf * kfr$ frames) on the map volume.

One of the main issues when using vSLAM to localize walking humanoids, is that the linear and vertical accelerations acting on the robot reduce the quality of image acquisition. To confront this problem, the pose graph correction pipeline selects keyframes for the graph (once every kfr frames), with the least amount of blurring. To quantify image blurriness, we use the variance of Laplacian method. The Laplacian is a common operator in image processing used to highlight regions of rapid pixel intensity changes, such as edges. If the variance of the image produced by the Laplacian operator is low, there are fewer edges and the image is blurry. Our algorithm selects as keyframes the frames with the least amount of blurriness, i.e. when their computed variance is higher than the user-defined blurring threshold (*blur_thr*). To select the default value of the *blur_thr*, we experimented with the trajectories of Table 5. Our experiments show that the blurred pictures have a Laplacian variance smaller than 80 ([4] presents similar results). We choose to set the default value equal to 100 to select sharper images.

Algorithm 1 illustrates the remaining of the pose correction pipeline. Table 2 explains the notations used in Algorithms 1 and 2. The voxel grid of the keyframe $k-kfr$ ($KF\text{-}VG_{l-1}$) is initialized with the global KF-VG (line 3). The global KF-VG is constructed by fusing the voxel grids of the frames between the $k-kfr$ and the frame $k-1$, using the estimated pose of the pose graph (RGB pose) for the frame $k-kfr$.

For each keyframe, we detect the ORB keypoints on the RGB image plane (line 4), and we compute the ORB descriptors for all 2D image coordinates of the detected keypoints $p_q = (u, v)^T \in \Omega$ [51].

Algorithm 1 Pose Graph Correction

Input: $image_{RGB}, GlobalKF\text{-}VG, T_{global}$
Output: $GlobalKF\text{-}VG, T_{global}, Voxelmap$
1: **for** each frame keyframe k, k>0 **do**
2: $l \leftarrow k/kfr$ // KF counter
3: KF-$VG_{l-1} \leftarrow GlobalKF\text{-}VG$ // init KF-VG of frame k-kfr
4: $featureDetection(image_{RGB}, ORB)$
5: $T_l \leftarrow RGBposeEstimation()$
6: **if** $k\%(kf * kfr) = 0$ **then**
7: $T_{opt} \leftarrow poseGraphOptimization()$
8: **for** $t \in [0, l-1]$ **do**
9: $FuseVGs$(GlobalKF-VG, KF-VG$_t$, T$_t$, t, k)
10: **end for**
11: $FuseVGs$(GlobalVG, GlobalKF-VG, T$_l$, l, k)
12: $T_{global} \leftarrow T_{opt} * T_{global}$
13: $Voxelmap \leftarrow raycast(GlobalKF\text{-}VG)$
14: **end if**
15: **end for**

The feature descriptors are then used to estimate point correspondences between consecutive keyframes. To accomplish this we match ORB keypoint pairs using the Fast Library for Approximate Nearest Neighbors (FLANN) [40]. The set of feature matches between two frames that are above a distance threshold, are formulated as constraints for global optimization, as described in Section 3.3.1. As an example, Figure 3 shows the matches of the ORB keypoints between two consecutive keyframes.

Given the extracted features of the RGB image, line 5 estimates the position of the keyframe and the pose T_l. The kf input parameter determines how often we correct the optimized pose of the system and update the Global VG. Thus, when the algorithm selects the kf most recent keyframes (line 6), we use the pose graph optimization to rectify all the estimated poses T_l and compute the optimized pose T_{opt} (line 7). This part is described in detail

Algorithm 2 Fuse Voxel Grids (FuseVGs)

Input: $dstVG, srcVG, T, t, k$
Output: $dstVG$
1: **for all** voxels $p(x, y, z)$ of the global VG **do**
2: $p_{src} \leftarrow T^{-1} * p$
3: $KF\text{-}TSDF_t(p) \leftarrow srcVG.interp(p_{src})$
4: update $TSDF_{k,t}(p)$ and $W_{k,t}(p)$ of $dstVG$ using eqns. 4, 5
5: **end for**

Table 2. Notations used in Algorithms 1 and 2

Algorithm 1	
GlobalVG	The Global 3D Voxel Grid (VG) is the baseline KinectFusion map representation (Figure 1) .
GlobalKF-VG	The Global KF Voxel Grid in which all the frames between consecutive keyframes are integrated (Figure 1).
$\boldsymbol{KF\text{-}VG_t}$	VG corresponding to the t-th keyframe. This VG is equal to the value of the $GlobalKF\text{-}VG$ as it has been configured until the execution of keyframe k.
$\boldsymbol{T_{global}}$	The estimated pose out of the KinectFusion pipeline.
$\boldsymbol{T_l}$	The estimated pose of keyframe l using the features of the RGB input images.
$\boldsymbol{T_{opt}}$	The optimized pose used in the correction of the T_{global} pose.
Algorithm 2	
srcVG	The input Voxel Grid, the information of which fused to $dstVG$.
dstVG	The output Voxel Grid for frame k, in which the $srcVG$ is integrated.
$\boldsymbol{T}$	The optimized pose of the keyframe to which the Voxel Grid $srcVG$ corresponds.
$\boldsymbol{KF\text{-}TSDF_t(p)}$	The result of the interpolation on $srcVG$ for voxel p used by the eqn. 4 to compute the corresponding voxel of the $dstVG$.
$\boldsymbol{TSDF_{k,t}(p)}$	The TSDF value of $dstVG$ for voxel p calculated by eqn. 4.
$\boldsymbol{W_{k,t}(p)}$	The weight value of $dstVG$ for voxel p calculated by eqn. 5.

in Section 3.3.1. Next, we integrate the VGs of all previous keyframes (line 9) and the Global KF-VG (line 11) into the Global VG. Section 3.3.2 explains the fusion between VGs. The KinectFusion pipeline for the next input frame $k + 1$ uses the updated pose T_{global} (line 12) and the 2D voxel map computed by the raycast algorithm (line 13).

3.3.1 ***Pose Graph Optimization.*** To rectify the estimated KinectFusion poses, we use pose graph optimization based on the detected RGB features. Every node in the graph corresponds to a keyframe pose, linked to the observations of the detected 2D features for that frame. Consecutive keyframes are connected by edges that model the error of the relative transformation between two frames against the global pose estimates from KinectFusion (T_{global}).

The module optimizes the graph and outputs a set of rectified poses which are used to align the preprocessed KF-VGs obtained earlier. For graph optimization we use the Levenberg-Marquardt method implemented by the open-source framework g^{2}o [33].

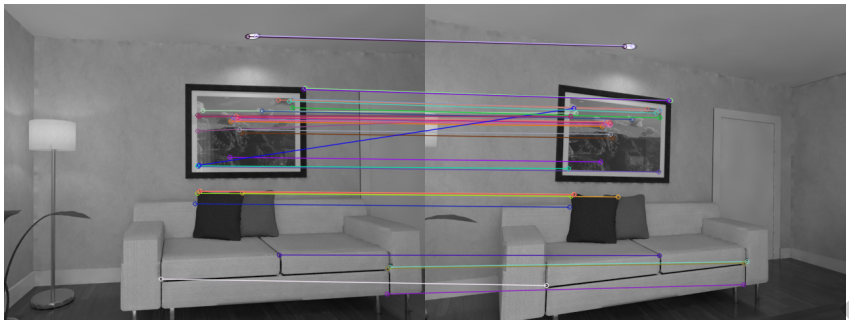


Fig. 3. ORB feature matches between consecutive keyframes (frames #50 and #100) of ICL-NUIM lr.kt2 trajectory (see Table 5 in Section 5.1 for a description of the input trajectories).

3.3.2 ***Updating the VGs.*** In the final block of the algorithm, the refined estimates obtained from pose graph optimization are integrated with the existing map representation. To accomplish this we maintain multiple voxel grids of the TSDF for each keyframe k (KF-VG). Defining the keyframe rate as kfr, keyframes are indexed as $k = \{0, kfr, 2 * kfr, ...\}$.

When new kf pose estimates are available from pose graph optimization (Algorithm 1, line 6), every $kf * kfr$ frames, we integrate the stored KF-VGs into the Global VG (**FuseVG**, Algorithm 2) using the refined pose estimates from optimization (T_l). Each destination voxel p needs to access a voxel cube (consisting of 8 voxels, as shown in Figure 2b) located in a random position p_{src}. This position depends on the optimization pose T (line 2). The trilinear interpolation between the cube vertices (line 3) is used to update the Global KF-VG in the following equations (line 4):

$$TSDF_{k,t}(p) = \begin{cases} \frac{W_{k-1}(p)*TSDF_{k-1}(p)+c*KF-TSDF_t(p)}{W_{k-1}(p)+c}, & t = 0. \\ \frac{W_{k,t-1}(p)*TSDF_{k,t-1}(p)+c*KF-TSDF_t(p)}{W_{k,t-1}(p)+c}, & t > 0. \end{cases} \tag{4}$$

$$W_{k,t}(p) = \begin{cases} W_{k-1,t} + c, & t = 0. \\ W_{k,t-1} + c, & 0 < t \leq l. \end{cases} \tag{5}$$

where $p(x, y, z)$ is the voxel coordinates, $KF\text{-}TSDF_t(p)$ denotes the TSDF of the t-th KF-VG, c is a constant value and $TSDF_{k,t}(p)$ and $W_{k,t}(p)$ are the TSDF and weight values of the Global VG at keyframe k, respectively.

3.4 Baseline PG-SLAM Performance and Motivation

Figure 4 shows the input/output of various stages of the PG-SLAM algorithm for a scene from trajectory *rb.traj0*. As an illustration of the comparative complexity of the PG-SLAM algorithm, the timeline patterns of Figure 5 show the execution time per frame for two of the benchmark trajectories running on a 4-core ARM Cortex-A53 CPU. Both the PG-SLAM and the baseline KinectFusion executions are shown (both are precise versions). The long spikes every $kfr * kf = 250$ frames are due to the periodic execution of the FuseVG kernel which integrates the voxel grids of the five intermediate keyframes to the global 3D voxel grid. The shorter spikes, every $kfr = 50$ frames, represent the periodic execution of feature detection/extraction and the construction of intermediate voxel grids for each keyframe. In Figure 5b, short spikes have varying frequencies because the intermediate frames do not satisfy the blurring threshold. In that case, these keyframes are omitted from being fused to the global 3D voxel grid. Note that even the non-spiked frames in PG-SLAM require higher execution time compared

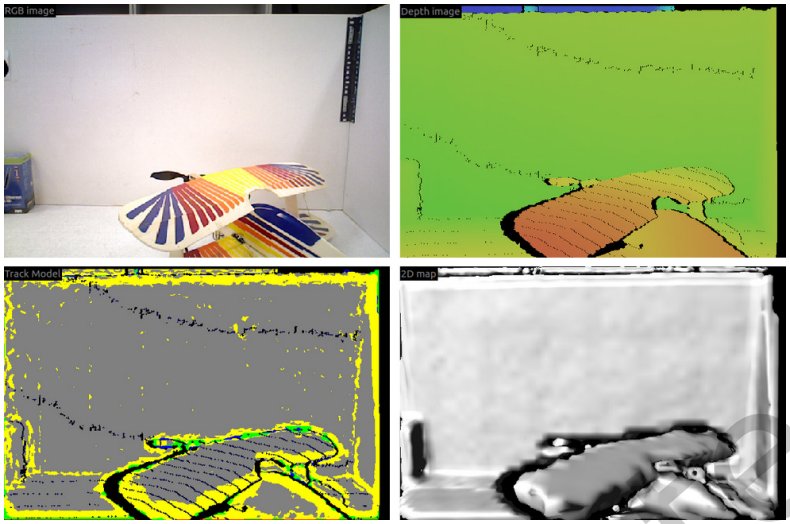


Fig. 4. The RGB scene (top left), the input depth frame (top right), the tracking output (bottom left), and the reconstructed 2D voxel map (bottom right). In tracking output, black corresponds to invalid NaN pixels from the depth sensor, and green to no correspondence between successive poses.

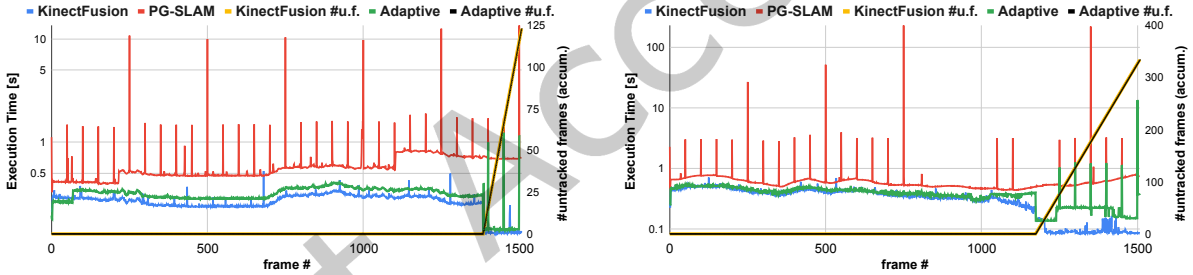


Fig. 5. The left y-axis (in logarithmic scale) shows the execution time of PG-SLAM vs. baseline KinectFusion on the (a) *rb.traj0* and (b) *lr.kt0* trajectories running on a multicore ARM (SW-only). The adaptive algorithm executes PG-SLAM only after the appearance of the first untracked frame. The right y-axis shows the accumulated number of untracked frames of KinectFusion and of the adaptive version of PG-SLAM. Note that pure PG-SLAM has zero untracked frames (to avoid clutter, the corresponding line is not shown in the figure).

with baseline KinectFusion. This is because of one additional invocation per frame of the integration kernel in PG-SLAM as shown in Figure 1.

Note that the baseline KinectFusion execution time drops considerably towards the end of the trajectory because ICP fails to track the frames and, thus, most frame processing is aborted. The characteristics of the trajectories, such as the increase in linear and rotational speed of the humanoid, poor scene illumination, and the blurring of the images may increase the deviation of the estimated pose from the actual pose. As more such poses deviate, the representation of the environment (3D Voxel Grid) used to track each subsequent frame will also

deviate from the real 3D representation, hence, pushing ICP to fail to track the next frame. As the Voxel Grid deviation accumulates (after a large number of frames), the probability of untracked frames goes up.

On the other hand, PG-SLAM tracks all frames successfully. The cumulative number of untracked frames in the right y-axis of Figure 5(a) and (b), motivates the deployment of PG-SLAM on humanoid robots as a robust vSLAM algorithm. As illustrated in Figure 1, the PG correction mechanism is applied periodically every $kfr * kf$ frames regardless of the presence of untracked frames. This, in turn, leads to periodic processing spikes, which can be clearly seen in Figure 5.

To mitigate the large processing overhead of the correction pipeline, we experimented with an adaptive version of PG-SLAM in which the correction mechanism is triggered only after the appearance of the first untracked frame. However, as shown in the timeline of Figure 5, this optimization does not avoid untracked frames despite the increased processing vs KinectFusion (after the first untracked frame). As we explained above, the reason is that due to the accumulated deviations in the 3D Voxel Grid, this just-in-time correction comes too late to have any positive effect.

4 DESIGN METHODOLOGY AND ARCHITECTURES

In this section, we describe our proposed MPSoC FPGA-based design methodology and the implementation of PG-SLAM architectures. The objective is to maximize the throughput of the most computationally demanding kernels with a selection of precise and approximate optimizations. We also schedule multiple tasks for concurrent execution using both the multicore CPU and the FPGA fabric resources to further increase the throughput of the PG-SLAM algorithm. At the same time, approximate PG-SLAM should not increase the number of untracked frames beyond a threshold after which tracking would not be able to resume.

4.1 Kernel-level optimizations

We experiment with three classes of kernel optimizations. *Precise* optimizations retain the accuracy of the baseline code. They include the usual assortment of loop-based transformations such as loop unrolling, loop pipelining, loop interchange to improve access locality, prefetching data from DRAM to the internal Block RAMs to reduce latency and instantiating multiple compute units to exploit task-level parallelism.

The PG-SLAM algorithm has a number of input parameters (shown in Table 3) that can be adjusted to trade-off the quality of the output with the running time and energy consumption of the program. *Approximate* optimizations, which may affect the accuracy of the baseline code, consist of two categories. In the first category (*ApproxA*), we relax the settings of the input parameters with respect to their default values (the range of values are shown in Table 3), whereas we use the precise implementations for all kernels. In the second category (*ApproxB*), we relax the settings of the input parameters and, at the same time, we use the approximate implementations of the kernels as shown in Table 4.

Note that we do not modify all PG-SLAM input parameters from their default value since this does not offer any interesting insights on the behavior of the algorithm. Specifically, we only relax the default settings of the following parameters: *tr, ir, icp, mu, pd0-pd2, kf, kfr, fts, blr_thr*.

Some widely explored approximations across most kernels are the following: (a) Using half precision 16-bit floating point format (5-bit exponent and 10-bit mantissa) instead of 32-bit float. Half precision fp16 arithmetic is widely used in our methodology aiming at lower resource requirements, thus, allowing more kernels to be implemented in the FPGA fabric. (b) Loop perforation, an approximate computing technique that skips loop iterations [56]. In dense point clouds used in vSLAM, loop perforation, when judiciously applied, does not cause degradation of output quality due to spatial locality of the input scenes and the reconstructed map.

Next, we discuss kernel-specific approximate optimizations as shown in Table 4.

Table 3. Input parameters of PG-SLAM, and their default (underlined) and approximate values.

Name	Range of Values	Description
	KinectFusion Parameters	
csr	320x240	Input frame resolution
tr	1, 2, 3 frames	Tracking rate
ir	1, 2, 3 frames	Integration rate
icp	1e-5, 0.01, 0.5	ICP threshold
vr	256^3 voxels	3D voxel grid dimensions
vm	8^3 m	Volume Map resolution
mu	0.1, 0.2, 0.3, 0.5, 0.6	Truncation distance used for voxel updates (integration)
pd0-pd2	pd0: 1,2,3,5,10 pd1: 0, 5 pd2: 0, 4	Max # of ICP iterations in each pyramid level
	Pose Correction Parameters	
kf	3, 5	Max # of keyframes
kfr	50, 100, 150, 200, 250	Keyframe rate
fts	100, 120, 150, 500	Max # features for detection
knn_dr	0.75	KNN distance for matching features between keyframes
fts_m	65	Min # feature matches betw. keyframes for features to be added in pose graph t
pgo_m	6	Max # iterations of pose graph optimization
blr_thr	100, 200, 300, 400, 500	Image blurring threshold

Bilateral Filter. (a) In some cases, the filter can be bypassed if it does not contribute much to pose tracking. (b) Instead of the default 5x5 array of coefficients used for convolution, we use a smaller 3x3 array, by peeling away the outer rows and columns of the 5x5 array. (c) We remove the range filtering stage of the bilateral filter to eliminate the invocation of an exponent function in the coefficient filter.

Integration. (a) We experiment with a variety of loop perforation approaches which include (i) skipping altogether the computation of new TSDF values, and using, instead, the old TSDF values (i.e., the values at the same position in the previous frame), (ii) copying the newly computed TSDF values of neighboring positions to the skipped positions. (b) We eliminate some of the branches that are used in the loops to check for special (typically error) conditions. (c) We use simpler functions or even fixed values to replace costly floating point operations that exhibit limited variance across loop iterations.

Raycast. (a) The inner loop of Raycast traverses a ray using steps of variable size, and the steps become smaller when approaching a surface in the voxel grid. Instead, we use steps of constant and larger size to achieve a deterministic schedule and size of voxel prefetching. (b) Similar to loop perforation, only a fraction of rays is traversed. (c) Trilinear interpolation accesses 8 different TSDF values for averaging (Figure 2b). We use simpler interpolation schemes that require fewer TSDF accesses and/or FP operations. We have extensively investigated the behavior of PG-SLAM for multiple combinations of voxel points in the trilinear interpolation and we concluded that using only two points (instead of eight) located the opposite side of the cube does not create large errors (Figure 2b). (d) We use fast math to approximate expensive floating point operations. (e) We perform Raycast at a lower rate, once every two frames.

Table 4. List of all approximate optimizations for each PG-SLAM kernel, and their default (underlined) and approximate values for the non-binary approximations.

Optim. Name	Description and Range of Values	Optim. Name	Description and Range of Values
Bilateral Filter		**FuseVG**	
BF_Off	Disable BF kernel invocations	**F_PGKf**	Pose graph correction using only the VGs of the last kf keyframes
BF_Coeff	Use 3x3 coefficients instead of 5x5		
BF_HP	Use fp16 arithmetic	**F_LP**	Loop perforation (Skip z-loop iter.) z-loop: <u>1</u>, 2, 4, 6
BF_Range	No range filter		
Tracking		**F_HP**	Use fp16 arithmetic
Tr_LP	Loop perforation (SW and HW opt.) x-loop: <u>1</u>, 2, 4 y-loop: <u>1</u>, 4, 8	**F_FPOp**	Eliminate expensive FP ops
		F_ULP	Unroll inner loop using the data of 1 input voxel at each iteration
Tr_HP	Use fp16 arithmetic		
Integration		**F_Cnst**	Eliminate the constant c in eq. 4 to reduce the # of mult/tions
Int_SLP	Loop perforation (Skip z-loop iter.)		
Int_CLP	Loop perforation (Copy) z-loop: <u>1</u>, 2, 3, 4, 6	**F_SkIn**	Skip the last kernel invocation (line 11, Alg. 1)
Int_HP	Use fp16 arithmetic	**F_TrInt**	Use fewer points for trilinear interpolation (2 instead of 8)
Int_Br	Eliminate checking		
Int_FPOp	Eliminate expensive FP operations		
Raycast		**Other**	
R_Step	Use larger ray steps <u>0.075</u>, 0.1125	**O_PGC**	Pose graph correction is called only if at least one of the last kfr frames is untracked (discussion in section 3.4)
R_LP	Skip computing of rays x-loop: <u>1</u>, 2, 4 y-loop: <u>1</u>, 2, 4, 8		
R_TrInt	Use fewer points for trilinear interpolation (2 instead of 8)		
R_Fast	Use –ffast-math		
R_Rate	Skip one frame		

All rays can be processed concurrently as separate threads as there is no data dependence between them. However, accessing the TSDF values of the voxel grid results in a non-contiguous and irregular memory access pattern, making memory bursting, data distribution to internal BRAMs and data reuse particularly challenging. Therefore, the Raycast kernel is scheduled for execution on the ARM processor since the CPU cache hierarchy performs better with irregular accesses than the hardware accelerator. All approximate techniques for Raycast refer to this software-based implementation.

FuseVG. To efficiently map the FuseVG component in the FPGA fabric, we split the computations of the kernel into two parts to proactively manage the irregular memory access pattern of the kernel. The first part (called FuseVG Interp. in Figure 6) is executed on the ARM processor concurrently with the Integration HW accelerator. Specifically, for each destination voxel, it reads the 8 voxels from the source VG and stores the result of the trilinear interpolation to the FuseVG input buffer (lines 2,3 of the Algorithm 2). The second part, the FuseVG HW accelerator, implements line 4 of the Algorithm 2 on HW, reading the input buffer row-wise.

A few more comments concerning approximate optimizations: (a) The initial pose graph correction algorithm used the VGs of all past keyframes (i.e. from the outset of the trajectory), and fused them to the global 3D voxel grid to correct the current pose. Since this is clearly inefficient both in terms of execution time and amount of storage, the *F_PGKf* approximation uses only the VGs of the last kf keyframes. (b) In *F_ULP*, which is a combination of loop unrolling with loop perforation, the inner loop is unrolled and the input value $kfTSDF_t(p)$ is being reused to compute $TSDF_{k,t}(p)$ (equation 4) for two successive positions. (c) The *F_TrInt* approximation is similar to *R_TrInt* and is applied to the FuseVG Interp. kernel to further reduce the amount of computations.

Other. We avoid the pose graph correction operation if there have been no untracked frames within the last *kfr* frames. The lack of untracked frames is a strong indication that the correction might be unnecessary at this point.

4.2 System-level scheduling

PG-SLAM consists of a large number of kernels whose data dependences may hinder their concurrent intra- and inter-frame execution. For example, the dependence from Raycast to tracking in KinectFusion (via the 2D voxel map) makes inter-frame kernel-level parallelism challenging. On the other hand, the pose graph correction pipeline can be executed in parallel to KinectFusion, even more so since its much lower execution rate at $\frac{1}{kfr}$ (or $\frac{1}{kf*kfr}$ for FuseVG) allows its very high execution overhead to be amortized across multiple frames.

FuseVG accesses the voxel grids (VGs) of the last kf keyframes and fuses them into the Global VG using trilinear interpolation accessing non-contiguous, irregular memory patterns which are based on the estimated RGB pose. Due to this irregularity, and since this interpolation operation can take place immediately after the formation of a keyframe, we execute the interpolation function on the ARM CPU. Figure 6 shows the system-level scheduling of three frames (regular, keyframe every $1 : kf$ frames, keyframe every $1 : kf * kfr$ frames). For brevity, we omit most of the kernels of the last keyframe. Note the CPU execution of FuseVG interpolations, as well as the concurrent execution of the Integr. B kernel (in HW) and the Raycast kernel (in CPU). Concurrency is possible because these two kernels access two different voxel grids, i.e. the KF-VG and the Global VG, respectively.

We leverage every opportunity for parallel execution at the system level as follows: (i) we use OpenMP to exploit intra-kernel parallelism. For example, each CPU core processes in parallel 25% of the Raycast rays. (ii) we exploit inter-task parallelism between CPU threads and hardware kernels. For example, Raycast executes in parallel with the IntegrationB kernel. Inter-task parallelism is always enabled when there are no inter-kernel dependencies.

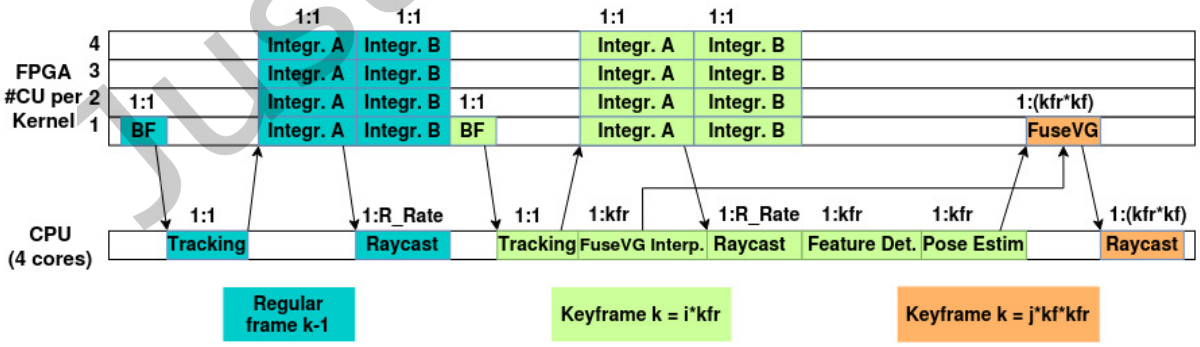


Fig. 6. System-level scheduling of PG-SLAM kernels.

5 EXPERIMENTAL EVALUATION

5.1 Methodology

For our study, we used the KinectFusion implementation of the SLAMBench suite [6, 13, 43], and we implemented the pose graph correction algorithm in C/C++ and OpenCV. The code was synthesized in Verilog using HLS tools and was implemented using the Vitis™ Unified Software Platform (v2019.2). We targeted the Xilinx UltraScale+ MPSoC ZCU102 Evaluation Kit whose FPGA includes a quad-core 1.2 GHz ARM Cortex-A53 processor paired with 4 GB DDR4 main memory and is based on Xilinx's 16nm FinFET+ programmable logic fabric. The FPGA fabric was clocked at 300 MHz in all our experiments. We measure power dissipation on the FPGA using the Power Management Bus (PMBus) protocol which monitors multiple power rails.

We use the trajectories of Table 5 in our experimental evaluation. The first type includes three trajectories generated when a NAO humanoid robot walks in the lab environment [69]. NAO was retrofitted with an Intel RealSense D455 RGB-D camera to provide the real-time RGB and depth images rb.traj[0-3] (Figure 7). The camera uses two $1280x720$ image sensors at 30 Hz, calculates depth information by using the disparity of the two camera images looking at the same scene, and provides the two synchronized and downscaled $320x240$ video streams (RGB and depth) to the SLAM pipeline. The second type includes four camera trajectories lr.kt[0-3] from ICL-NUIM, a synthetic dataset providing RGB-D sequences from a living room model [28]. Even though this dataset is not from a walking robot, we chose to include it in our experiments to evaluate the robustness of PG-SLAM in different settings. For example, our experiments show that the algorithm works well even with trajectory *lr.kt3*, which is typically excluded from evaluation of conventional SLAM algorithms due to the large number of untracked frames.

Table 5. Input trajectories used for evaluation. #untracked frames refer to precise KinectFusion and PG-SLAM SW executions.

Name	Type	# of frames	#untracked frames		% NaN Pixels
			KFusion	PG-SLAM	
rb.traj0	Walking robot	1508	123	0	6.7%
rb.traj1		2158	0	0	7.3%
rb.traj2		1469	0	0	8.9%
lr.kt0	Living Room	1510	336	0	0%
lr.kt1		967	0	0	0%
lr.kt2		882	0	0	0%
lr.kt3		1242	24	0	0%

As an output quality metric, we use the number of frames that could not be tracked by the ICP tracker. Columns 3 and 4 of Table 5 indicate the number of untracked frames when the trajectory is used as input to precise KinectFusion and PG-SLAM algorithms, respectively. Note that the precise PG-SLAM succesfully eliminates all untracked frames. Column 5 shows the percentage of invalid pixels in the depth frame acquired by the RGB-D camera (see Section 5.3 for an in-depth discussion of NaN pixels).

An open source release of our implementations and data can be found in [1].

5.2 Results

5.2.1 ***Performance Analysis.*** Our first experiment shows the performance and the speedup of the fastest precise and the fastest approximate (individual) accelerators compared with the baseline (i.e. non-optimized and precise) SW and HW implementations (Table 6). The fastest approximate accelerators enable all the approximations of

Fig. 7. The NAO humanoid used in our experiments.

Table 4 for each kernel without compromising the system's ability to recover from untracked frames (that was our constraint). For example, the performance of the unoptimized HW kernel of the Bilateral filter is 0.56 invocations / sec, whereas the fastest precise implementation yields 613.5x speedup, and approximate optimizations further increase the speedup to 824x. The Bilateral filter and the Integration kernel are characterized by regular memory access patterns (2D and 3D, respectively), and readily available loop-level parallelism and they exhibit very high performance improvement for the fastest approximate case.

Table 6. Performance and speedup of kernel implementations (N = 1 accelerators) running the *rb.traj0* trajectory. The Baseline HW performance for FuseVG is too low to be considered in practical implementations and is set equal to 0.

	SW-only	**Baseline HW**	**Fastest and Precise**		**Fastest and Approximate**		
			Speedup wrt.		Speedup wrt.		
	Hz	Hz	SW-only	Baseline HW	SW-only	Baseline HW	Fastest Precise
Bilateral (SW)	18.1	–	–	–	21.7x	–	–
Bilateral (HW)	–	0.56	18.98x	613.5x	25.5x	824x	1.34x
Tracking (SW)	48.6	–	–	–	10.5x	–	–
Tracking (HW)	–	3.41	1.12x	16.03x	4.34x	61.9x	3.86x
Integration (SW)	5.3	–	–	–	3.52x	–	–
Integration (HW)	–	1.04	1.77x	9.05x	5.96x	30.4x	3.4x
FuseVG (SW)	0.75	–	–	–	37.6x	–	–
FuseVG (HW)	–	0	14.6x	∞	46x	∞	3.15x
Raycast (SW)	10.9	–	–	–	10.3x	–	–

Table 6 shows that the HW approximate implementations of Bilateral filter, Integration and FuseVG kernels have a much higher speedup compared to the SW implementations. For example, FuseVG fastest approximate implementation yields 37.6x speedup in SW and 46x in HW. Based on the results of Table 6, our PG-SLAM platforms

always map these three kernels on the FPGA fabric. Moreover, we vary the parameters and approximation policies (knobs) shown in Tables 3 and 4 to define a large space of PG-SLAM HW/SW configurations, both precise and approximate, which span the performance vs. accuracy space.

Based on the contribution of each kernel to execution time of the fastest precise PG-SLAM configuration in Figure 9, and the results of Table 6, we produce a plethora of approximate PG-SLAM circuits. We evaluate each such circuit for all combinations of the values of the input parameters of Table 3, and we generate the scatter plot of Figure 8. Note that changing the values of the input parameters of PG-SLAM does not require us to build new hardware circuits, so this phase is very fast. This plot shows the performance (frames/sec) vs. the accuracy (number of untracked frames), for a set of the most promising PG-SLAM designs running the *rb.traj0* trajectory. Note that real-time PG-SLAM processing (i.e. at least 30 fps) is only possible only through *ApproxB* configurations since the fastest precise configuration reaches only 2.3 fps, and the fastest *ApproxA* configuration reaches 3.6 fps.

Configurations C1 and C2 correspond to the two edges of the Pareto frontier, with C1 being the fastest approximate design (36.04 fps at a cost of 473 untracked frames), and C2 the fastest approximate with no untracked frames running at 28.2 fps. Table 7 shows the settings for configurations C1 and C2.

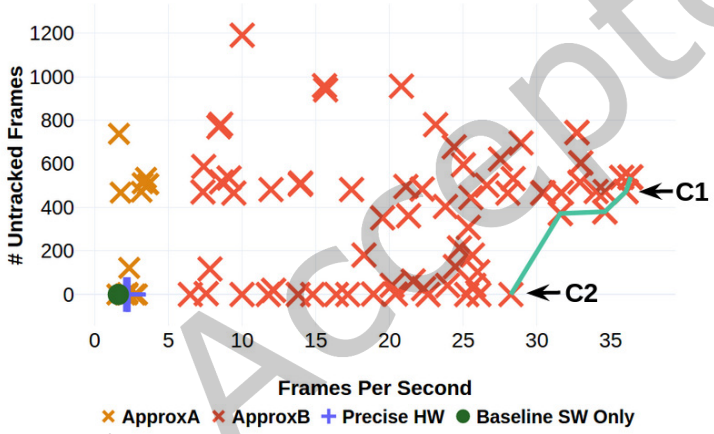


Fig. 8. Throughput vs. number of untracked frames of various PG-SLAM configurations running the *rb.traj0* benchmark. *ApproxA* and *ApproxB* configurations are defined in Section 4.1. The green line marks the Pareto frontier. The Precise HW mark corresponds to a configuration in which all HW kernels are executed precisely. In each case, Raycast, Tracking, FuseVG interpolation and Feature Detection are executed in the ARM CPU and all remaining kernels in HW.

Table 7. C1 and C2 configurations of Figure 8

Conf.	Approximations & Input Parameters
C1	BF, Integration, FuseVG in HW. All HW approximations enabled. Only *F_ULP* is off. Tracking, Raycast in SW. $kfr = 150, fts = 150, pgo\_m = 1, icp = 0.01, pd0 - pd2 = (2, 0, 0)$
C2	Same approximations as C1 except that *F_Cnst*=off, *O_PGC*=on $kfr = 100, icp = 0.01, pd0 - pd2 = (3, 0, 0)$

Our evaluation for the remaining trajectories of Table 5 reveals that the performance of their corresponding fastest precise configuration is in the range 1.68 fps–2.3 fps, and of the fastest approximate configuration 19.7 fps– 37.8 fps. Performance for all trajectories of Table 5 is discussed more thoroughly in Section 5.2.3.

As a general rule, loop perforation provides substantial speedup for almost all kernels without noticeable loss of accuracy, provided that the number of skipped iterations is small. Since the speed of the humanoid is relatively low (especially along the z-axis) compared with the input frame rate, most of the updates of the 3D voxel grids can be copied from previous versions of the grids.

FuseVG is an interesting case, since its baseline HW execution which occurs when the trilinear interpolation executes in HW after we invoke the FuseVG, is prohibitively slow. As we mentioned in Section 4.2, the execution of the FuseVG interpolation is scheduled on the ARM CPU concurrently with the execution of non-keyframes, effectively hiding the FuseVG interpolation overhead. The approximate FuseVG kernel is 46x faster than the SW-only implementation. For the tracking kernel, skipping the two higher pyramid levels and/or reducing the maximum iterations in each level is very beneficial for performance. Finally, as we mentioned in Section 4.1, the Raycast kernel is not suitable for hardware acceleration mainly due to complex memory accesses in the large 64 MB 3D voxel grid. We have also evaluated the possibility of merging the Integration and Raycast kernels in a single accelerator given that these two kernels are executed in sequence and communicate through the 3D voxel grid. This approach did not work since the Integration kernel does not update the TSDF values in the same order as these are accessed by the Raycast kernel, requiring to store a large number of partial values. For these reasons, the Raycast kernel is executed as a standalone component on the ARM CPU in all implementations.

An illustration of the relative contribution of each kernel to execution time is shown in Figure 9. Note the much higher execution time required for a keyframe (esp. for the keyframes for which pose graph optimizations are invoked every $kf * kfr$ frames) owing to the execution of the FuseVG and PGC-FuseVG kernels. For regular frames (i.e. not keyframes), Raycast and tracking emerge as a bottleneck in fast hardware solutions, since they are always scheduled on the CPU for execution. The contribution of the FuseVG kernel to the total execution time becomes less pronounced in approximate solutions. This is due to the positive effects of executing the FuseVG interpolation much earlier as discussed in Section 4.1. A similar effect can be observed for the approximate Raycast kernel whose execution fully overlaps with the IntegrationB kernel.

Optimizing for RMSE. The number of untracked frames has been our evaluation metric in all experiments thus far. In Table 8, we expand our evaluation to include the effects of approximations on the Root Mean Square Error (RMSE) between the ground truth and the estimated trajectories of the agent for the ICL-NUIM benchmarks (for which the ground truth is available [28]). Since PG-SLAM is optimized for trajectories of a walking humanoid, it suffers (as expected) from a higher per frame average RMSE compared with precise KinectFusion. Note that we run the PG-SLAM algorithm using the approximate configurations that result in the lowest average RMSE for the lr.kt benchmarks (and not with the C1 and C2 configurations). Table 8 indicates that the RMSE of approximate PG-SLAM can still be close to the RMSE of precise KinectFusion, without compromising real-time performance requirements.

5.2.2 ***Significance Analysis.*** Since the knobs presented in Tables 3 and 4 introduce a very large design space, our analysis needs to provide a relative ranking of the significance (impact) of each knob to the execution time of the PG-SLAM FPGA implementation. The relative impact of PG-SLAM optimizations on performance is quantified using Lasso [65], a regularized linear regression method that builds a model for coefficient selection aiming at minimizing the following loss function:

$$J(\Theta) = J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m}(\sum_{i=1}^{n}(\Theta^T X_i - y_i)^2 + \lambda \sum_{j=1}^{m} |\theta_j|)$$

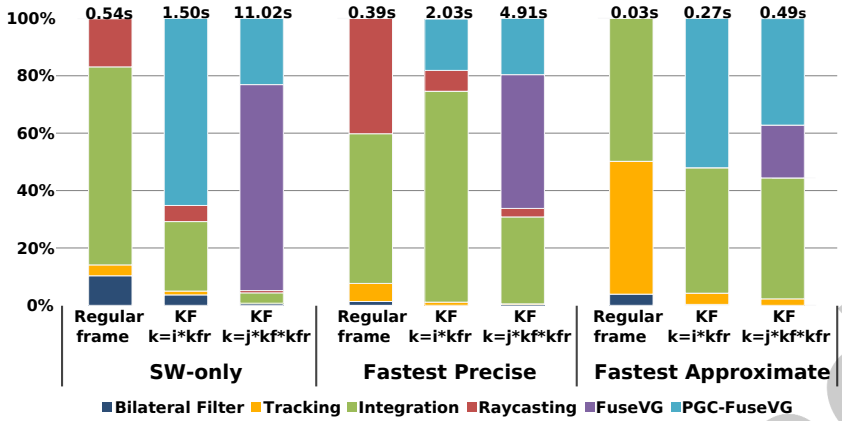


Fig. 9. Contribution of each PG-SLAM kernel to average execution time / frame for each one of the three types of frames. Average execution time is shown at the top of each bar. PGC-FuseVG refer to all kernels of the Pose Graph Correction pipeline (Figure 1) except FuseVG. The Raycast and FuseVG Interp. executions are not shown in the fastest approximate configuration because they are executed concurrently with the Integr. B kernel. Fastest Approximate refers to configuration C1.

Table 8. Performance and RMSE of the PG-SLAM approximate designs and RMSE of precise KinectFusion for the ICL-NUIM benchmarks. The number of untracked frames is zero for PG-SLAM designs.

Input	PG-SLAM		Precise KinectFusion
	Performance [fps]	Avg. RMSE [meters]	Avg. RMSE [meters]
lr.kt0	22.70	0.0234	0.8454
lr.kt1	27.11	0.0502	0.0277
lr.kt2	23.36	0.0462	0.0207
lr.kt3	26.48	0.3271	0.3082

where Θ is the vector of the m coefficients to be computed, the vector X_i contains the knob settings and y_i is the resulting execution time for the i_{th} (out of a total of n) experiment. The result of this analysis are the coefficients θ_j for each knob j, which represent the relative contribution of this knob to the execution time. The regularization parameter λ controls the amount of variance of the model. The Lasso method tends to favor the case when there is a small number of significant optimizations by pushing the Θ coefficients of insignificant optimizations to zero.

Our prior work indicates that precise optimizations are higher ranked compared with approximations, and consistently contribute to higher performance without degrading accuracy [26]. For example, *Loop Interchange* and *Loop Pipeline* of the Integration kernel, as well as *Loop Pipeline* of the Tracking and *Loop Unroll* of the BF kernels are the most impactful precise optimization of the KinectFusion algorithm according to [26]. Thus, in our analysis, we always enable all precise optimizations, since they will be part of any optimized PG-SLAM configuration. We use all approximations of Table 4 and the parameters *tr, ir, icp, mu, pd0-pd2, kf, kfr, fts, blr_thr* as features in Lasso regression. Table 9 shows the Lasso analysis for the PG-SLAM execution time. The table reports features up to the second degree since these provide a better trade-off between model accuracy (lower Mean Square Error, MSE) and model complexity.

Table 9. Lasso Ranking of PG-SLAM performance. Only the coefficients with magnitude at least 0.01 are shown in the Table.

Combined (MSE=0.0038)			
Feature	**Coeff.**	**Feature**	**Coeff.**
R_LP^2	0.502	kfr	-0.029
Tr_LP^2	0.076	pd*icp	0.026
F_LP^2	0.057	F_HP^2	0.013
Int_SLP	-0.052	BF_Off	-0.011
$Int\_NCU\_HP^2$	-0.052	F_LP*F_HP	-0.010
pd^2	0.047	BF_Off*R_Rate	-0.010
icp	0.039		

Lasso ranking validates our previous observation that loop perforation is the most impactful approximations, esp. for raycast and tracking. Moreover, the approximations of FuseVG and Integration have major effect on performance due to their significant contribution to the total execution time. The feature *Int_NCU_HP* (not shown in Table 4) denotes that the *Int_HP* approximation is impactful only when applied in multiple integration accelerators.

Table 10 shows how Lasso ranking is used to gradually apply sets of optimizations according to their impact on performance. The baseline HW configuration executes PG-SLAM at 0.25 fps using unoptimized HW accelerators for the Bilateral filter, the Integration and fuseVG kernels, and executing the Tracking and Raycast kernels on the ARM CPU. By including all precise optimizations, the throughput increases to 2.31 fps (9.24x faster than the baseline). As we move to approximate optimizations (which are less impactful than the precise ones), the speedup wrt. to the previous configuration drops to 5.8x for approximations with $|\text{coeff}| \geq 0.05$, and to 1.8x for approximations with $0.05 > |\text{coeff}| \geq 0.01$. The remaining 11 approximations (not shown in Table 9) contribute only an additional 1.48x performance improvement. The last two rows show the resource utilization and the average power dissipation (separately for the PL fabric and CPU) for each configuration. Note that PL power dissipation increases as the resource utilization goes up. Similarly, Table 11 shows the effect of the most impactful optimizations (when these optimizations are applied individually) on performance, resource utilization and power dissipation.

Table 10. Optimization selection based on Lasso ranking. Clusters of more impactful optimizations have larger speedup compared with less impactful optimizations.

	Baseline (BL)	**Fastest Precise**	**Approx. $\|\text{coeff}\| \geq 0.05$**	**Approx. $\|\text{coeff}\| \geq 0.01$**	**Fastest Approx. (C1)**
Throughput (fps)	0.25	2.31	13.45	24.28	36.04
Speedup wrt. BL	1	9.24	53.8	97.1	144.2
Speedup wrt. previous config. (to the left)	-	9.24	5.8	1.8	1.48
% Resource Utilization (BRAM, DSP, FF, LUT)	(2,2,6,8)	(17,33,40,63)	(34,37,42,70)	(33,36,33,60)	(34,31,31,55)
Average Power (PL, CPU) in Watts	(0.87,1.2)	(2.49, 1.34)	(3.12, 1.39)	(2.87, 1.38)	(2.68, 1.31)

Table 11. Impact of individual approximations with |coeff| ≥ 0.05.

	Baseline (BL)	Fastest Precise	R_LP	Tr_LP	F_LP	Int_SLP
Throughput (fps)	0.25	2.31	2.75	2.36	2.54	2.58
Speedup wrt. BL	1	9.24	11	9.44	10.16	10.3
% Resource Utilization (BRAM, DSP, FF, LUT)	(2,2,6,8)	(17,33,40,63)	(17,33,40,63)	(17,33,40,63)	(17,33,40,63)	(17,33,40,62)
Average Power (PL, CPU) in Watts	(0.87, 1.2)	(2.49, 1.34)	(2.51, 1.32)	(2.48, 1.34)	(2.49, 1.34)	(2.45, 1.36)

5.2.3 ***Timeline Analysis.*** The timelines of Figure 10 show, for each trajectory, the execution time per frame for the fastest precise configuration, as well as the two configurations C1 and C2 that correspond to the two edges of the Pareto frontier (similar to Figure 8). Approximations substantially raise performance from around 2 fps to a minimum 19.7 fps (*rb.traj2*) and a maximum 37.8 fps (*lr.kt3*). In most cases, the execution profiles of C1 and C2 are very similar. The exception is in *rb.traj0* due to the large number of untracked frames (frame #751 to #1223) of configuration C1. Aggressive approximations in C1 push the ICP to lose track of a continuous stream of frames, yet even in that case, PG-SLAM is able to recover and continue successfully.

Note that for a long sequence of frames in *lr.kt* trajectories, the PG-SLAM implementations do not invoke the pose graph correction stage and there are no spikes in their timeline. This is due to the lower RGB image quality of *lr.kt* trajectories (compared with *rb.traj*), which disqualifies most frames to be selected as keyframes, since their Laplacian variance is lower than *blurr_thr* (Section 3.3).

The two approximate configurations in *lr.kt3* set the integration rate *ir* equal to 2, which causes large execution time variations between successive frames. Note also the "high frequency" variations, which are present in all timelines, and are almost entirely due to the small intra-frame execution time variations of the Tracking kernel.

Note that the fastest approximate designs C1 for each trajectory are different. For example, in *rb.traj0*, the *F_Const* approximation is enabled, *pd0-pd2* = (2,0,0) and *kfr=150*. For the *lr.kt3* case, the *F_Const* approximation is disabled, *pd0-pd2* = (2,0,0) and *kfr=250*, whereas $tr = ir = 2$. For brevity, we do not present the exact settings of the approximate configurations for each trajectory.

5.2.4 ***Resource Utilization and Power Dissipation.*** Figure 11 shows the FPGA resource utilization for the fastest precise and for configuration C1 when running the *rb.traj0* trajectory. Note that only three kernels are implemented in the FPGA fabric (Integration has 4 instances in C1), whereas all the remaining kernels are executed in the ARM CPU. BF kernel approximations drastically reduce the resources used by this kernel, thus allowing the instantiation of multiple integration units.

It is interesting to note that LUT utilization reaches only 62.78% and 55.15% on the precise and approximate configurations, respectively. Ideally, the remaining resources would be used to implement more types of kernels, and/or to instantiate more compute units per kernel on the FPGA fabric. However, the software implementation of Raycast, Tracking, feature detection/extraction, etc. is faster than any hardware implementation, even if the latter is approximated. Moreover, adding extra Integration/Bilateral/FuseVG units has negligible effect on performance. For example, the configuration that also implements the Tracking kernel on the FPGA fabric reaches only 25 fps at a LUT utilization of 72.46%.

Configuration C1 dissipates on average 2.68W on the PL FPGA fabric, and 1.31W on the 4-core ARM CPU. The numbers for the fastest precise configuration are 2.49W and 1.34W, respectively. These power numbers are also shown in Table 10.

5.2.5 ***Feature Detectors.*** Feature detection is a low-level image processing operation used to detect features in an image (typically in the form of edges, corners, lines, etc.) so that these features can be tracked in subsequent images. The selection of feature detectors is a critical decision which affects both the performance and the number of detected features in vision-based applications. In this section, we evaluate the efficiency of multiple feature detector algorithms such as ORB, BRISK, KAZE, SURF and SIFT on the metrics of interest of PG-SLAM. [61] includes a thorough description of these algorithms and evaluation of their performance and accuracy on the problem of image registration.

Figure 12 shows that when the ORB detector is used in feature matching between keyframes, PG-SLAM has the lowest number of untracked frames, and, in most cases, the highest performance. This is true regardless of the software/hardware configuration (baseline SW and fastest approximate are shown in Figure 12). For example, using the KAZE feature detector when the C1 configuration runs the *rb.traj0* trajectory, reduces performance from 36.04 fps (ORB detector) down to 20.6fps as shown in Figure 12c. ORB is based on the FAST corner detector (well-known for its good performance), while KAZE uses a scale-normalized determinant of Hessian Matrix on multiple scale levels. Note also that aggressive approximations greatly reduce the importance of feature descriptor selection in terms of accuracy, as shown in Figure 12d. For all the reasons above, we selected the ORB feature detector for all platform configurations. The findings of our evaluation are compatible to the analysis of [61].

5.3 Depth sensor noise

The quality of the depth images provided by the RGB-D camera may suffer from limited accuracy and stability due to depth holes and inconsistent depth values [37]. Depth holes typically occur when the camera cannot determine the real depth of an object. This can happen near the edges of objects, as well as within smooth and shiny surfaces such as mirrors, or when objects lie outside of the depth range (0.4m to over 10m for the Intel RealSense camera). When the depth of a pixel cannot be estimated, the camera produces a default value, which is typically NaN (Not a Number) (Figure 13). The last column of Table 5 shows the average percentage of NaN pixels across all frames in the trajectory.

Since the presence of NaN pixel values in the camera output may negatively affect the numerical stability and the performance of the SLAM pipeline, we have experimented with two NaN mitigation techniques. The first approach is to replace individual NaN pixels as they are generated by the camera with (i) a default constant depth value, or (ii) interpolated values (e.g. applying bilinear interpolation), or (iii) the output of morphological filters such as dilation and erosion. As a second, alternative approach, we let the NaN pixels propagate inside the pipeline and tag them with a default value at the **depth2vertex** and **vertex2normal** functions, which are used to prepare the input to the tracking function (Section 3). This default value is a signal for the ICP tracker to ignore/bypass these pixels when checking for convergence.

Extensive experimentation indicated that the second method exhibits the lowest number of untracked frames for all *rb.traj* trajectories (the *lr.kt* trajectories have no NaN pixels). Since invalid depth pixels tend to appear in large clusters, the first method has an adverse effect on the number of invalid pixels since any operation between a valid pixel and a NaN value results into a NaN value.

6 CONCLUSION

In this paper, we introduced PG-SLAM, an extension to KinectFusion algorithm that helps a humanoid robot recover its pose when the baseline KinectFusion tracking system fails. The pose graph optimization phase is very computationally expensive limiting real-time PG-SLAM performance. Hence, we proposed numerous approximations to reduce computation requirements and we explored the architecture of multiple HW/SW FPGA designs that implement approximate versions of PG-SLAM. We showed that even though approximations are necessary to achieve real-time operation, they need to be applied carefully to avoid large numbers of untracked

frames. Our fastest approximate FPGA design achieved 37.8 fps (31 fps for the design with zero untracked frames), compared with 1.87 fps of the fastest precise HW-based implementation and 1.59 fps of the SW-only precise implementation on the ARM CPU. An interesting outcome of our work was to illustrate that efficient FPGA implementations of real-life applications like SLAM often require complex interactions between CPU execution of SW components and execution of hardware accelerators.

REFERENCES

[1] Oct 2021. https://github.com/csl-uth/PG-SLAM_fpga. DOI: 10.5281/zenodo.5616787.

[2] Mohamed Abouzahir, Abdelhafid Elouardi, Rachid Latif, Samir Bouaziz, and Abdelouahed Tajer. 2018. Embedding SLAM Algorithms: Has it come of age? *Robotics and Autonomous Systems* 100 (2018), 14 – 26.

[3] Nikolay A. Atanasov, J. L. Ny, Kostas Daniilidis, and George J. Pappas. 2015. Decentralized active information acquisition: Theory and application to multi-robot SLAM. *IEEE International Conference on Robotics and Automation (ICRA)* (2015), 4775–4782.

[4] Raghav Bansal, Gaurav Raj, and Tanupriya Choudhury. 2016. Blur image detection using Laplacian operator and Open-CV. *2016 International Conference System Modeling & Advancement in Research Trends (SMART)* (2016), 63–67.

[5] Paul J. Besl et al. 1992. A Method for Registration of 3-D Shapes. *IEEE Trans. Pattern Analysis and Machine Intelligence* 14, 2 (1992).

[6] Bruno Bodin, Harry Wagstaff, Sajad Saeedi, Luigi Nardi, Emanuele Vespa, John Mawer, Andy Nisbet, Mikel Luján, Steve B. Furber, Andrew J. Davison, Paul H. J. Kelly, and Michael F. P. O'Boyle. 2018. SLAMBench2: Multi-Objective Head-to-Head Benchmarking for Visual SLAM. *CoRR* abs/1808.06820.

[7] Konstantinos Boikos and Christos-Savvas Bouganis. 2016. Semi-dense SLAM on an FPGA SoC. In *26th International Conference on Field Programmable Logic and Applications, (FPL), Lausanne, Switzerland, August 29 - September 2, 2016*, Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele (Eds.). IEEE, 1–4.

[8] Konstantinos Boikos and Christos Savvas Bouganis. 2017. A High-Performance System-on-Chip Architecture for Direct Tracking for SLAM. In *27th International Conference on Field Programmable Logic and Applications, (FPL), Ghent, Belgium, September 4-8*. 1–7.

[9] Konstantinos Boikos and Christos Savvas Bouganis. 2019. A Scalable FPGA-Based Architecture for Depth Estimation in SLAM. In *15th International Symposium on Applied Reconfigurable Computing, (ARC) Darmstadt, Germany, April 9-11*. 181–196.

[10] Vanderlei Bonato, Eduardo Marques, and George A. Constantinides. 2009. A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots. *J. Signal Process. Syst.* 56, 1 (2009), 41–50.

[11] Guillaume Bresson, Z. Alsayed, Li Yu, and S. Glaser. 2017. Simultaneous Localization and Mapping: A Survey of Current Trends in Autonomous Driving. *IEEE Transactions on Intelligent Vehicles* 2 (2017), 194–220.

[12] David J. Bruemmer and Mark S. Swinson. 2003. Humanoid Robots. In *Encyclopedia of Physical Science and Technology (Third Edition)* (third edition ed.), Robert A. Meyers (Ed.). Academic Press, New York, 401–425.

[13] Mihai Bujanca, Paul Gafton, Sajad Saeedi, Andy Nisbet, Bruno Bodin, Michael F. P. O'Boyle, Andrew J. Davison, Paul H. J. Kelly, Graham D. Riley, Barry Lennox, Mikel Luján, and Steve B. Furber. 2019. SLAMBench 3.0: Systematic Automated Reproducible Evaluation of SLAM Systems for Robot Vision Challenges and Scene Understanding. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*. IEEE, 6351–6358.

[14] 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.

[15] Brian Curless and Marc Levoy. 1996. A Volumetric Method for Building Complex Models from Range Images.. In *23rd Annual Conference on Computer Graphics and Interactive Techniques, (SIGGRAPH), New Orleans, LA, USA, August 4-9, 1996*. 303–312.

[16] Igor Cvisic, Josip Cesic, Ivan Marković, and I. Petrović. 2018. SOFT-SLAM: Computationally efficient stereo visual simultaneous localization and mapping for autonomous unmanned aerial vehicles. *J. Field Robotics* 35 (2018), 578–595.

[17] A. Davison, I. Reid, N. Molton, and O. Stasse. 2007. MonoSLAM: Real-Time Single Camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29 (2007), 1052–1067.

[18] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 2018. SuperPoint: Self-Supervised Interest Point Detection and Description. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 224–236.

[19] Jakob Engel, Thomas Schöps, and Daniel Cremers. 2014. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8690)*. Springer, 834–849.

[20] Weikang Fang, Yanjun Zhang, Bo Yu, and Shaoshan Liu. 2017. FPGA-based ORB feature extraction for real-time visual SLAM. In *International Conference on Field Programmable Technology, (FPT), Melbourne, Australia, December 11-13, 2017*. 275–278.

[21] Marcel Flottmann, Marc Eisoldt, Julian Gaal, Marc Rothmann, Marco Tassemeier, Thomas Wiemann, and Mario Porrmann. 2021. Energy-efficient FPGA-accelerated LiDAR-based SLAM for embedded robotics. In *International Conference on Field-Programmable Technology, (FPT) Auckland, New Zealand, December 6-10, 2021*. IEEE, 1–6.

[22] Christian Forster, Simon Lynen, L. Kneip, and D. Scaramuzza. 2013. Collaborative monocular SLAM with multiple Micro Aerial Vehicles. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2013), 3962–3970.

[23] Quentin Gautier, Alric Althoff, and Ryan Kastner. 2019. FPGA Architectures for Real-time Dense SLAM. In *30th IEEE International Conference on Application-specific Systems, Architectures and Processors, (ASAP), New York, NY, USA, July 15-17, 2019*. 83–90.

[24] Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. 2014. Real-time 3D reconstruction for FPGAs: A case study for evaluating the performance, area, and programmability trade-offs of the Altera OpenCL SDK. In *2014 International Conference on Field-Programmable Technology, FPT Shanghai, China, December 10-12, 2014*. 326–329.

[25] A. Gil, Ó. Reinoso, M. Ballesta, and Miguel Juliá. 2010. Multi-robot visual SLAM using a Rao-Blackwellized particle filter. *Robotics Auton. Syst.* 58 (2010), 68–80.

[26] Maria Rafaela Gkeka, Alexandros Patras, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas. 2021. FPGA Architectures for Approximate Dense SLAM Computing. In *24th Conference on Design, Automation and Test in Europe (DATE) Virtual Conference, February 1-3, 2021*.

[27] Mengyuan Gu, Kaiyuan Guo, Wenqiang Wang, Yu Wang, and Huazhong Yang. 2015. An FPGA-based real-time simultaneous localization and mapping system. In *International Conference on Field Programmable Technology, (FPT) Queenstown, New Zealand, December 7-9, 2015*. 200–203.

[28] Ankur Handa et al. 2014. A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In *ICRA Hong Kong, China, May*.

[29] A. Hornung, Kai Wurm, and Maren Bennewitz. 2010. Humanoid robot localization in complex indoor environments. *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, 1690 – 1695. https://doi.org/10.1109/IROS.2010.5649751

[30] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time 3D reconstruction and interaction using a moving depth camera. *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 559–568.

[31] Olaf Kähler et al. 2015. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices. *IEEE Trans. Vis. Comput. Graph.* 21, 11 (2015).

[32] C. Kerl, Jürgen Sturm, and D. Cremers. 2013. Dense visual SLAM for RGB-D cameras. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2013), 2100–2106.

[33] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. 2011. G2o: A general framework for graph optimization. *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 3607 – 3613. https://doi.org/10.1109/ICRA.2011.5979949

[34] Jinyu Li, Bangbang Yang, Danpeng Chen, Nan Wang, Guofeng Zhang, and Hujun Bao. 2019. Survey and evaluation of monocular visual-inertial SLAM algorithms for augmented reality. *Virtual Real. Intell. Hardw.* 1 (2019), 386–410.

[35] Haomin Liu, Mingyu Chen, Guofeng Zhang, Hujun Bao, and Yingze Bao. 2018. ICE-BA: Incremental, Consistent and Efficient Bundle Adjustment for Visual-Inertial SLAM. In *IEEE Conference on Computer Vision and Pattern Recognition, (CVPR), Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 1974–1982.

[36] Runze Liu, Jianlei Yang, Yiran Chen, and Weisheng Zhao. 2019. eSLAM: An Energy-Efficient Accelerator for Real-Time ORB-SLAM on FPGA Platform. In *56th Annual Design Automation Conference, (DAC), Las Vegas, NV, USA, June 02-06, 2019*. ACM, 193.

[37] Tanwi Mallick, Parthapratim Das, and Arun Majumdar. 2014. Characterizations of Noise in Kinect Depth Images: A Review. *IEEE Sensors Journal* 14, 6 (2014), 1731–1740.

[38] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4 (2016).

[39] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. 2003. FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges. In *IJCAI*.

[40] Marius Muja and David Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. *VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications* 1, 331–340.

[41] Raul Mur-Artal et al. 2015. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Trans. Robotics* 31, 5 (2015).

[42] Raul Mur-Artal and J. D. Tardós. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics* 33 (2017), 1255–1262.

[43] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham D. Riley, Nigel P. Topham, and Stephen B. Furber. 2015. Introducing SLAMBench, a Performance and Accuracy Benchmarking Methodology for SLAM. In *International Conference on Robotics and Automation, (ICRA), Seattle, WA, USA, 26-30 May*. 5783–5790.

[44] J. Nikolic, J. Rehder, M. Burri, Pascal Gohl, Stefan Leutenegger, P. Furgale, and R. Siegwart. 2014. A synchronized visual-inertial sensor system with FPGA pre-processing for accurate real-time SLAM. In *International Conference on Robotics and Automation, (ICRA), Hong Kong, China, May 31 - June 7, 2014*.

[45] Jinwook Oh, Jungwook Choi, Guilherme C. Januario, and Kailash Gopalakrishnan. 2016. Energy-Efficient Simultaneous Localization and Mapping via Compounded Approximate Computing. In *IEEE International Workshop on Signal Processing Systems (SiPS), Dallas, TX, USA, October 26-28, 2016.*

[46] Giuseppe Oriolo, Antonio Paolillo, Lorenzo Rosa, and Marilena Vendittelli. 2012. Vision-based Odometric Localization for humanoids using a kinematic EKF. In *12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012).* 153–158.

[47] R. Ozawa, Y. Takaoka, Y. Kida, K. Nishiwaki, J. Chestnutt, J. Kuffner, J. Kagami, H. Mizoguch, and H. Inoue. 2005. Using visual odometry to create 3D maps for online footstep planning. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 3. 2643–2648 Vol. 3.

[48] Yan Pei, Swarnendu Biswas, Donald S. Fussell, and Keshav Pingali. 2019. SLAMBooster: An Application-Aware Online Controller for Approximation in Dense SLAM. In *28th International Conference on Parallel Architectures and Compilation Techniques, (PACT), Seattle, WA, USA, September 23-26, 2019.*

[49] Yan Pei, Swarnendu Biswas, Donald S. Fussell, and Keshav Pingali. 2020. A Methodology for Principled Approximation in Visual SLAM. In *International Conference on Parallel Architectures and Compilation Techniques (PACT), Virtual Event, GA, USA, October 3-7, 2020.* 373–386.

[50] Stylianos Piperakis, Nikolaos Tavoularis, Emmanouil Hourdakis, Dimitrios Kanoulas, and Panos Trahanias. 2019. Humanoid Robot Dense RGB-D SLAM for Embedded Devices. https://doi.org/10.13140/RG.2.2.13008.76803

[51] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: an efficient alternative to SIFT or SURF. *Proceedings of the IEEE International Conference on Computer Vision*, 2564–2571.

[52] Sajad Saeedi et al. 2018. Navigating the Landscape for Real-Time Localization and Mapping for Robotics and Virtual and Augmented Reality. *Proc. IEEE* 106, 11 (2018), 2020–2039.

[53] Renato F. Salas-Moreno, Richard A. Newcombe, H. Strasdat, P. Kelly, and A. Davison. 2013. SLAM++: Simultaneous Localisation and Mapping at the Level of Objects. *2013 IEEE Conference on Computer Vision and Pattern Recognition* (2013), 1352–1359.

[54] Patrik Schmuck. 2017. Multi-UAV collaborative monocular SLAM. *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017), 3863–3870.

[55] Thomas Schöps, Torsten Sattler, and M. Pollefeys. 2019. BAD SLAM: Bundle Adjusted Direct RGB-D SLAM. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), 134–144.

[56] Stelios Sidiroglou-Douskos et al. 2011. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *ESEC/FSE, Szeged, Hungary, Sept. 2011.*

[57] Edgar Simo-Serra, Eduard Trulls, Luis Ferraz, Iasonas Kokkinos, Pascal Fua, and Francesc Moreno-Noguer. 2015. Discriminative Learning of Deep Convolutional Feature Point Descriptors. In *2015 IEEE International Conference on Computer Vision, ICCV Santiago, Chile, December 7-13, 2015.* 118–126.

[58] Randall C. Smith, Matthew Self, and Peter C. Cheeseman. 1987. Estimating uncertain spatial relationships in robotics. *Proceedings. 1987 IEEE International Conference on Robotics and Automation* 4 (1987), 850–850.

[59] Amr Suleiman et al. 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).

[60] Y. Takaoka, Y. Kida, S. Kagami, H. Mizoguchi, and T. Kanade. 2004. 3D map building for a humanoid robot by using visual odometry. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, Vol. 5. 4444–4449 vol.5. https://doi.org/10.1109/ICSMC.2004.1401231

[61] Shaharyar Ahmed Khan Tareen and Zahra Saleem. 2018. A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK. *International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)* (2018), 1–10.

[62] Daniel Tortei Tertei, Jonathan Piat, and Michel Devy. 2014. FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig14, Cancun, Mexico, December 8-10, 2014.* IEEE, 1–6.

[63] Daniel Tortei Tertei, Jonathan Piat, and Michel Devy. 2016. FPGA design of EKF block accelerator for 3D visual SLAM. *Comput. Electr. Eng.* 55 (2016), 123–137.

[64] Simon Thompson, Satoshi Kagami, and Koichi Nishiwaki. 2006. Localisation for Autonomous Humanoid Navigation. In *2006 6th IEEE-RAS International Conference on Humanoid Robots.* 13–19.

[65] R. Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society (Series B)* 58 (1996).

[66] Carlo Tomasi and Roberto Manduchi. 1998. Bilateral Filtering for Gray and Color Images. In *6th International Conference on Computer Vision (ICCV), Bombay, India, January 4-7, 1998.*

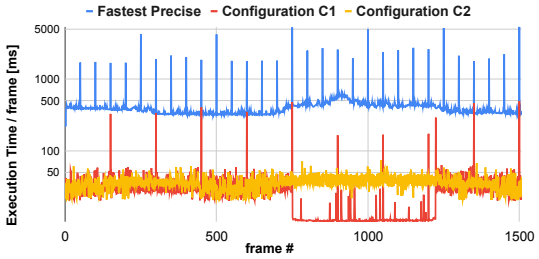
[67] Bastien Vincke, Abdelhafid Elouardi, and Alain Lambert. 2012. Real time simultaneous localization and mapping: towards low-cost multiprocessor embedded systems. *EURASIP J. Embed. Syst.* 2012 (2012), 5.

[68] Thomas Whelan, Stefan Leutenegger, Renato F. Salas-Moreno, B. Glocker, and A. Davison. 2015. ElasticFusion: Dense SLAM Without A Pose Graph. In *Robotics: Science and Systems.*

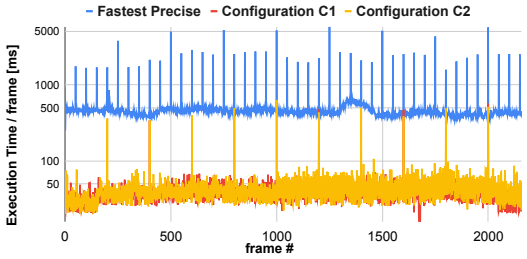
[69] Émilie Wirbel, Bruno Steux, Silvere Bonnabel, and Arnaud de La Fortelle. 2013. Humanoid robot navigation: From a visual SLAM to a visual compass. In *10th IEEE International Conference on Networking, Sensing and Control, ICNSC 2013, Evry, France, April 10-12, 2013*. 678–683.

[70] Zhilin Xu, Jincheng Yu, Chao Yu, Hao Shen, Yu Wang, and Huazhong Yang. 2020. CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, (FCCM), Fayetteville, AR, USA, May 3-6, 2020*. 33–37.

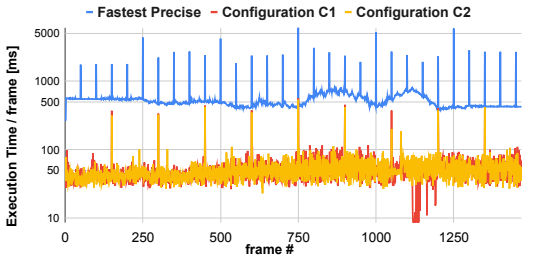
[71] Tianwei Zhang, Emiko Uchiyama, and Yoshihiko Nakamura. 2018. Dense RGB-D SLAM for Humanoid Robots in the Dynamic Humans Environment. In *IEEE-RAS International Conference on Humanoid Robots, Humanoids. Beijing, China, November 6-9, 2018*. 270–276.



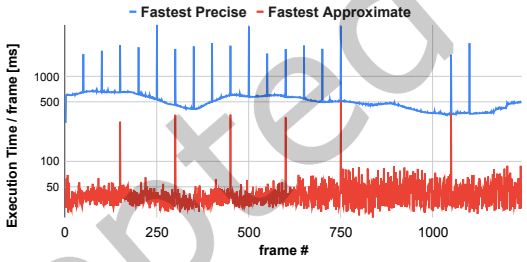
(a) *rb.traj0*. Prec:(2.3, 0), C1:(36.04, 473), C2:(28.2, 0)



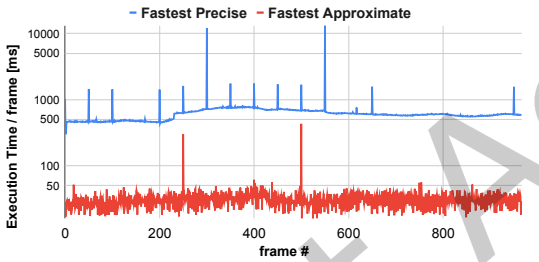
(b) *rb.traj1*. Prec:(2.03, 0), C1:(26.6, 8), C2:(23.3, 1)



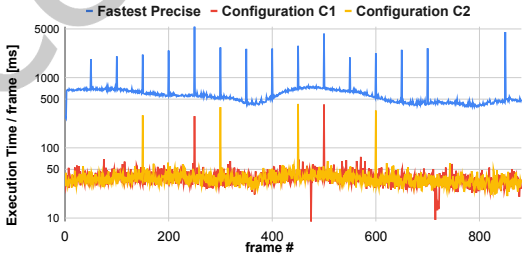
(c) *rb.traj2*. Prec:(1.78, 0), C1:(19.7, 115), C2:(19.5, 47)



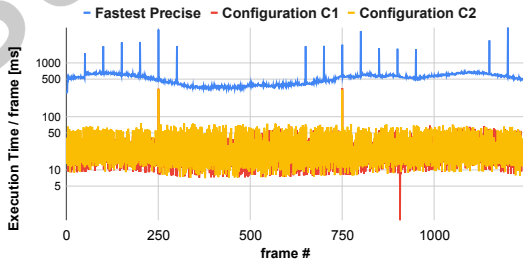
(d) *lr.kt0*. Prec:(1.8, 0), C1:(22.7, 0)



(e) *lr.kt1*. Prec:(1.6, 0), C1:(31.8, 0)



(f) *lr.kt2*. Prec:(1.68, 0), C1:(26.5, 8), C2:(26, 0)



(g) *lr.kt3*. Prec:(1.87, 0), C1:(37.8, 2), C2:(31, 0)

Fig. 10. Execution time/frame of the fastest precise (blue), fastest approximate regardless of the number of untracked frames C1 (red), and fastest approximate with the lowest number of untracked frames C2 (orange) for each input trajectory. For the trajectories in (d) and (e), C1 and C2 coincide. Each configuration pair depicts fps and # of untracked frames. Y-axis is in logarithmic scale.

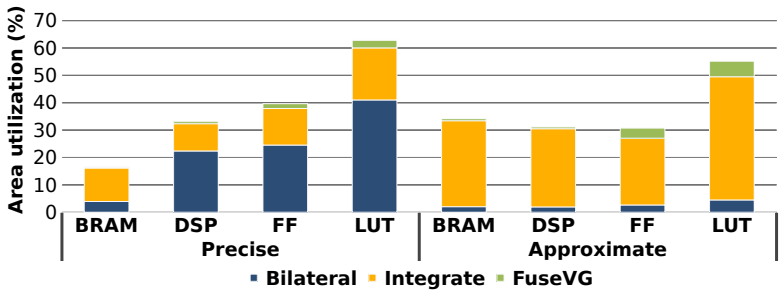
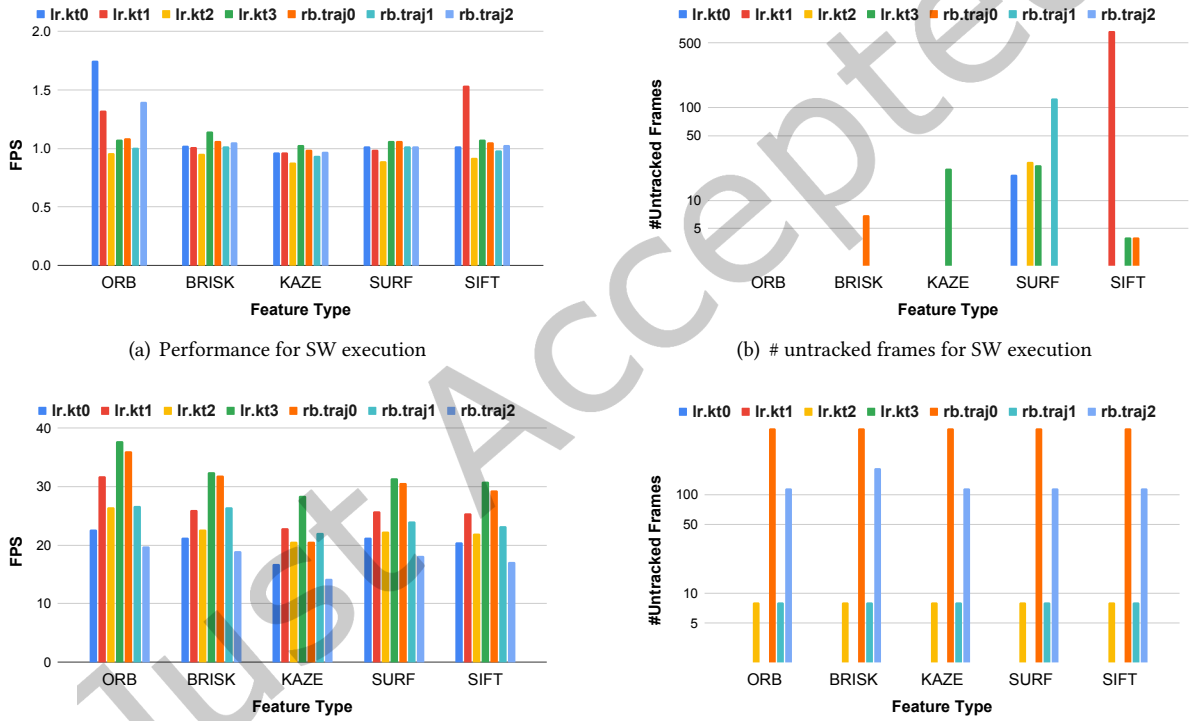


Fig. 11. FPGA resource utilization for the fastest precise and the fastest approximate (C1 in Figure 8) PG-SLAM implementations.



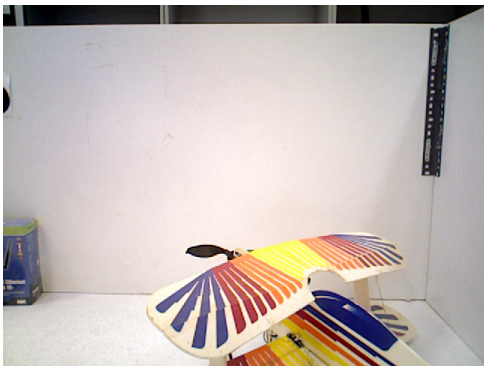
(a) Performance for SW execution

(b) # untracked frames for SW execution

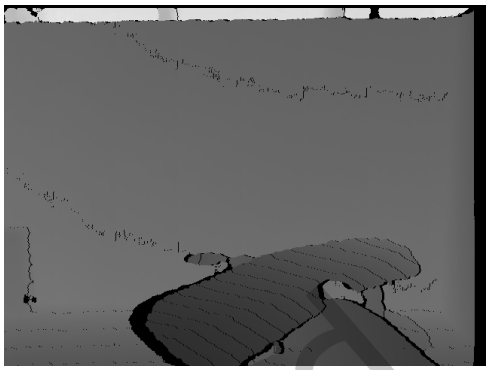
(c) Performance for C1 configurations

(d) # untracked frames for C1 configurations

Fig. 12. The effect of each feature descriptor on performance (fps) and # untracked frames for each trajectory. The C1 configurations may be different for each trajectory. # untracked frames axis is in logarithmic scale.



(a)



(b)

Fig. 13. (a) RGB frame #50 of rb.traj0. (b) The Depth frame #50 has 5% NaN pixels. They are drawn in black.