UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Programme of MSc Studies

Visual SLAM implementation and optimization on FPGAs

Master's Thesis

Maria Rafaela Gkeka

Supervisor: Nikolaos Bellas

Computer Systems Laboratory (CSL)

Volos 2020

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Programme of MSc Studies

Visual SLAM implementation and optimization on FPGAs

Master's Thesis

Maria Rafaela Gkeka

Supervisor: Nikolaos Bellas

Computer Systems Laboratory (CSL)

Volos 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Πρόγραμμα Μεταπτυχιακών Σπουδών

Υλοποίηση και βελτιστοποίηση αλγορίθμου visual SLAM σε FPGAs

Μεταπτυχιακή Διπλωματική Εργασία

Μαρία Ραφαέλα Γκέκα

Επιβλέπων: Νικόλαος Μπέλλας

Computer Systems Laboratory (CSL)

Βόλος 2020

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos D. Antonopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 16-10-2020

Acknowledgements

I would like to express my deepest gratitude to my supervisor Prof. Nikolaos Bellas for his guidance and invaluable contribution. I am also grateful to my advisors, Prof. Christos D. Antonopoulos and Prof. Spyros Lalis for their extremely useful help, ideas and feedback.

I would also like to thank Alexandros Patras for our collaboration over the last year.

Last but not least, I would like to thank people that were close to me. My friends, for the fun moments we spent together, their support and understanding throughout these years. My parents and my sister, for their encouragement and unwavering support.

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE Project VipGPU: Very Low Power GPUs for Mobile Robotics and Virtual Reality applications (project code: T1EDK-01149)

DISCLAIMER ON ACADEMIC ETHICS
AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Maria Rafaela Gkeka
16-10-2020

Abstract

In the field of robotic automation, Simultaneous Localization and Mapping (SLAM) is the problem of constructing and continuously updating a map of an unknown environment while keeping track of an agent's trajectory within this environment. SLAM is widely used in robotics, navigation, and odometry for augmented and virtual reality. Owing to the proliferation of cameras and the availability of new high performance and low power platforms, visual SLAM is rapidly becoming a popular domain in embedded computing with many different possible applications. An interesting visual SLAM category is dense SLAM which constructs and updates the map at pixel granularity at a very high computational and energy cost especially when operating under real-time constraints.

In this thesis, KinectFusion (a well-known dense visual SLAM algorithm) is approximated without exceeding user-defined error constraints. This is important to ensure that these approximations do not prevent the agent from navigating correctly in the environment. Specifically, this work introduces and evaluates a plethora of MPSoC FPGA designs for KinectFusion, featuring a variety of optimizations and approximations, to highlight the interplay between SLAM performance and accuracy.

We show that these approximations when applied on top of aggressive (but precise) hardware optimizations, can achieve almost real-time performance and higher energy efficiency compared with a software-only CPU implementation without compromising agent tracking and map construction.

Περίληψη

Η οικογένεια αλγορίθμων SLAM περιγράφει το πρόβλημα αυτόνομης κίνησης των ρομπότ. Συγκεκριμένα, αφορά την κατασκευή και τη συνεχή ενημέρωση του χάρτη ενός άγνωστου περιβάλλοντος, ενώ παράλληλα παρακολουθείται η πορεία κίνησης ενός πράκτορα μέσα στο ίδιο περιβάλλον. Το SLAM χρησιμοποιείται ευρέως στη ρομποτική, την πλοήγηση και την οδομετρία για εφαρμογές επαυξημένης ή εικονικής πραγματικότητας. Λόγω της ευρύας ανάπτυξης και χρήσης των καμερών, το οπτικό (visual) SLAM καθίσταται ένας δημοφιλής τομέας στα ενσωματωμένα υπολογιστικά συστήματα με πολλές διαφορετικές εφαρμογές. Μια ενδιαφέρουσα κατηγορία visual SLAM είναι το dense SLAM, όπου ο χάρτης κατασκευάζεται και ενημερώνεται κάνοντας χρήση όλης της πληροφορίας εισόδου από τον αισθητήρα. Επιπλέον, χαρακτηρίζεται από πολύ υψηλό υπολογιστικό και ενεργειακό κόστος, ειδικά όταν λειτουργεί υπό συνθήκες ανταπόκρισης του συστήματος σε πραγματικό χρόνο.

Σε αυτή τη διατριβή, ερευνάται η απόδοση του KinectFusion, ενός dense visual SLAM αλγορίθμου. Εφαρμόζονται βελτιστοποιήσεις της απόδοσης του συστήματος που μειώνουν την ακρίβειά του (προσεγγιστικές), χωρίς όμως η μείωση να υπερβαίνει τους περιορισμούς σφάλματος που καθορίζει ο χρήστης. Η διακύμανση του σφάλματος εντός συγκεκριμένων ορίων διασφαλίζει την σωστή πλοήγηση του πράκτορα στο περιβάλλον. Συγκεκριμένα, η εργασία αναλύει και αξιολογεί πληθώρα υλοποιήσεων του αλγορίθμου KinectFusion σε MPSoC FPGA, που εφαρμόζουν ποικίλες προσεγγιστικές, και μη, βελτιστοποιήσεις, ώστε να τονίσει την αλληλεπίδραση μεταξύ απόδοσης και ακρίβειας ενός SLAM συστήματος.

Επιπλέον, δείχνουμε ότι αυτές οι προσεγγίσεις όταν εφαρμόζονται σε συνδυασμό με αποδοτικές και ακριβείς βελτιστοποιήσεις υλικού, μπορούν να επιτύχουν απόδοση σχεδόν πραγματικού χρόνου και ενεργειακή απόδοση υψηλότερη από εκείνη που απαιτείται για την εκτέλεση σε έναν επεξεργαστή, ενώ παράλληλα διασφαλίζεται η σωστή παρακολούθηση του πράκτορα και η κατασκευή του χάρτη.

Table of contents

List of figures

List of tables

Abbreviations

wrt.	with respect to
e.g.	for example
SW	Software
HW	Hardware
FPGA	Field Programmable Gate Arrays
SoC	Systen-on-Chip
MPSoC	Multi-Processor System-on-Chip
RMSE	Root Mean Square Error
MSE	Mean Square Error
fps	frames per second
FP	Floating Point
HP	Half Precision
TSDF	Truncated Signed Distance Function
2D	Two-Dimensional
3D	Three-Dimensional

Chapter 1

Introduction

The proliferation of autonomous robots and unmanned aerial vehicles (UAVs) has created the need to construct highly accurate maps of their observed environment and to track the position and the trajectory of these agents within these maps. This process, which is referred to as Simultaneous Localization and Mapping (SLAM), typically merges data from various sensors such as stereo/mono and RGB-D cameras, laser scanners (lidars) and Inertial Measurement Units (IMUs) and involves a non-trivial amount of data processing.

Given that most robotic platforms have size, weight and energy limitations, SLAM is usually implemented using resource-constrained and power-efficient embedded systems. In order for such implementations to be useful they have to deliver high performance in a power constrained environment.

Due to the energy and performance limitations, most embedded visual SLAM implementations focus on *sparse* SLAM algorithms, which reduce computational requirements by maintaining only a sparse selection of key feature points and are typically limited to localization only. On the other hand, *dense* SLAM, which uses all pixels of the input frame for map reconstruction, provides the potential for richer 3D scene modeling. However, its high computational and energy requirements make real-time implementation very challenging, especially for an embedded system. Chapter 2 provides more information on dense and sparse SLAM aglorithms.

In an effort to tackle this challenge, we explore the large design space of FPGA-based dense SLAM accelerators, by combining application-specific optimizations with approximate computing. Approximate computing has been shown to accelerate computations and reduce energy requirements in various application domains [8]. In this thesis, we leverage

a set of algorithmic optimizations to study the trade off between performance and accuracy of localization and mapping in an effort to improve the design space exploration of similar applications.

1.1 Problem statement & Contributions

For our study, we use the KinectFusion algorithm of the SLAMBench suite [9]. Starting from the baseline C++/OpenMP implementation, we develop precise as well as approximate hardware accelerators for each of the most important components of the algorithm, by re-writing the code and using HLS directives. As a target platform we use the Ultrascale+ ZCU102 MPSoC board, and exploit the OpenCL API of Vitis™ Unified Software Platform [10] to generate and evaluate different solutions at a high level of abstraction.

KinectFusion is a closed-loop algorithm which continuously embeds new information (depth frames) to a partially constructed 3D view of a globally consistent map. Therefore, the scale of approximations is limited by its impact on the convergence to a consistent view. Approximate computing provides a speedup of up to $9.4x$ compared with the precise FPGA implementation without violating the tight constraints on the cumulative trajectory error.

Our contributions are summarized as follows:

- We enumerate a list of precise and approximate optimizations targeting KinectFusion, and, based on those, we introduce a multitude of parameterizable FPGA implementations spanning the performance vs. accuracy space.
- We evaluate these implementations to provide quantitative and qualitative analysis of the effect of each optimization separately and groups of optimizations in tandem on the performance and accuracy of KinectFusion running on an MPSoC FPGA.
- Finally, we provide a mechanism to rank the significance of each optimization on performance and accuracy for each kernel and for the whole application, and we show how this mechanism drives the generation of FPGA configurations with user-defined performance requirements.

1.2 Overview of the content

This thesis is organized in five chapters as follows:

Chapter 2 describes the concepts and terminologies used in SLAM systems. It also provides information about the Xilinx Zynq UltraScale+ MPSoC device used in our experiments and discusses related work.

Chapter 3 presents and analyzes the precise and approximate optimizations we apply on each kernel separately.

Chapter 4 presents the experimental results, corresponding to our work contributions. In particular, it provides the performance improvements achieved by our designs concerning error constraints and analyzes the significance of the optimizations described in chapter 3.

Chapter 5 concludes the thesis.

Chapter 2

Background

This chapter describes the background of SLAM computing and FPGA platforms. We start with a review, in Section 2.1, of visual SLAM algorithms characteristics and information about KinectFusion, the algorithm we implement and optimize on hardware. In Section 2.2 we describe the functionality of the FPGA platform, and in Section 2.3 we discuss prior work on FPGA-based implementations of SLAM algorithms.

2.1 SLAM and KinectFusion

2.1.1 SLAM Systems

Simultaneous Localization and Mapping (SLAM), a concept used to solve a very important problem in mobile robotics, consists of two parts:

- **mapping**, which refers to building a map of the agent's environment, and
- **localization**, which refers to the navigation of the environment using the map while keeping track of the agent's relative position and orientation.

Using visual input from a camera, SLAM algorithms estimates the path on which the camera has moved, and the positions in 3D space of all of the objects and features in an unknown environment that the camera has observed.

A robot to achieve autonomous navigation without any human input, needs to be able to localize itself in its environment based on the constructed map. Robot localization requires sensory information regarding its position and orientation of the robot within the built map.

Because of the significant role of input sensors in a SLAM system's localization part, the following subsection is an overview of the available types and their characteristics.

Types of input sensors

Sensors are divided into two main categories:

1. **Exteroceptive sensors**, which provide absolute position measurements, such as cameras and lasers

2. **Interoceptive sensors**, which generate relative position measurements, such as wheel odometers and IMUs (Inertial Measurement Units)

One of the demands on an autonomous robot is the ability to sense its environment. Proper sensor selection is crucial as it affects the quality and quantity of environmental information available to the robot and subsequently determines what SLAM approach is most suitable to be used [11]. The three major types of sensors applied to current SLAM technology are the following.

1. **Acoustic sensors**: *Sonar sensors* are mostly used underwater. However, the monotony of subsea regions means sonar depth information is much harder to interpret with high angular uncertainty. *Ultrasonic sensors* are generally the cheapest available source of spatial sensing for mobile robots. They are compatible with most surface types, whether metal or non-metal, clean or opaque, as long as the surface measured has sufficient acoustic reflectivity.

2. **Laser Range Finders**: Due to the high speed and high precision of laser range finders that enable them to produce highly accurate distance measurements, laser-based systems can obtain robust results in both indoor and outdoor environments.

3. **Visual sensors**: We will analyze the three basic types of sensors and the characteristics of the systems that use them.

    - *Monocular cameras* are modified refracting telescopes used to magnify the images of distant objects by passing light through a series of lenses and usually prisms [12]. They generate 2D images and used in SLAM systems with strict cost, area, and energy requirements (e.g. in mobile telephony). However, due to

the lack of clear depth information from a 2D image, a drawback is that the algorithms and software necessary for monocular SLAM are far more complex. Another weakness are the difficulties due to pure rotational motion. Since the estimation of 3D points are essential, it is unavoidable that a system based on monocular sensor will encounter numerical issues during periods of pure rotational motion or slow motion [13]. However, there exist works which can handle this situation, as described in [14]. LSD-SLAM [15], ORB-SLAM [16] and ORB-SLAM2 [17] are some noted SLAM systems which provide implementations based on Monocular camera sensor input.

- *Stereo cameras*. A stereo camera system consists of two cameras separated by a fixed distance. It can estimate depth by computing the difference between the two slightly offset camera images. Specifically, as figure 2.1 shows, owing to their disparate locations, a point in the physical world is projected differently by two cameras onto two film frames. In the right camera picture, the point in the left camera picture is shifted by a defined distance. The depth value can then be obtained if the relative location of each point in each camera is known. A good estimation of depth needs a decent baseline distance: The wider the baseline, the better the depth estimation. However, since the cameras are mounted on the robot, the distance between them can be limited. Some Stereo-based SLAM systems are ORB-SLAM2 [17], PL-SLAM [18] and StereoScan [19].

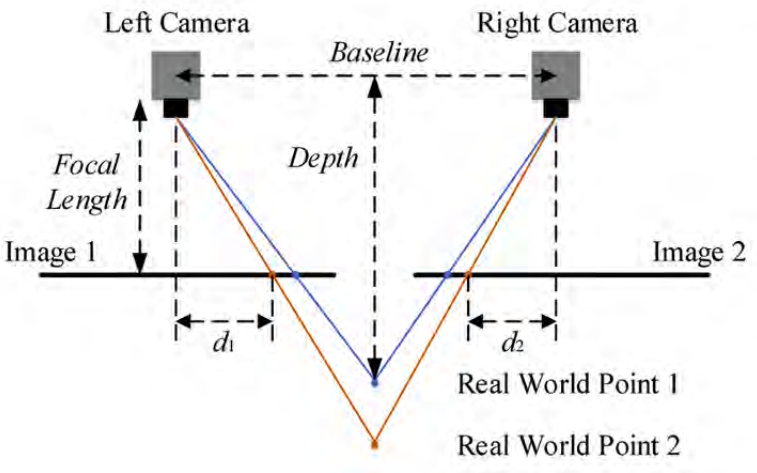

Figure 2.1: Stereo camera placement

- *RGB-D cameras*, such as Kinect [20] have high mobility and low cost. This

type of sensors are widely used in various applications while capture RGB images along with pixel-wise depth information. The most noted SLAM algorithms which use this sensor type are KinectFusion [21], ElasticFusion [22], BundleFusion [23], DVO-SLAM [24], Kintinuous [25] and InfiniTAM [26].

Visual Odometry

Odometry is used to estimate the sequential changes of sensor positions over time using sensors such wheel encoder to acquire relative sensor movement. Localization is the main task for autonomous vehicles to be able to track their paths and properly detect and avoid obstacles. According which one of the visual sensors referred above the autonomous machine use, this determines the type of *Visual Odometry*. Each camera-based odometry called Visual Odometry (VO). Vision-based odometry is one of the robust techniques used for vehicle localization [27]. VO can also be used in conjunction with information from other sources such as GPS, inertia sensors, wheel encoders, etc.

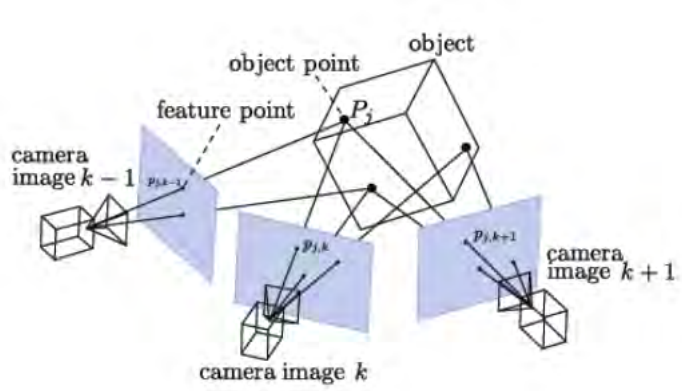

Figure 2.2: Visual SLAM pose estimation

Visual SLAM

Visual SLAM is a SLAM system which uses visual input sensors such as monocular cameras, stereo rigs, RGB-D cameras etc. The majority of modern visual SLAM systems are based on tracking a set of points through successive camera frames and use them to estimate the camera pose. These tracks are also used to triangulate their 3D position to create the map. Figure 2.2 shows three consecutive input frames where each one sees the same object from a different point of view. All input frames contain common features related to the object

characteristics, which the algorithm recognizes them approximately and uses them to estimate each frame camera pose and also fuse this information in the map. More information on this procedure will be found in section 2.1.2.

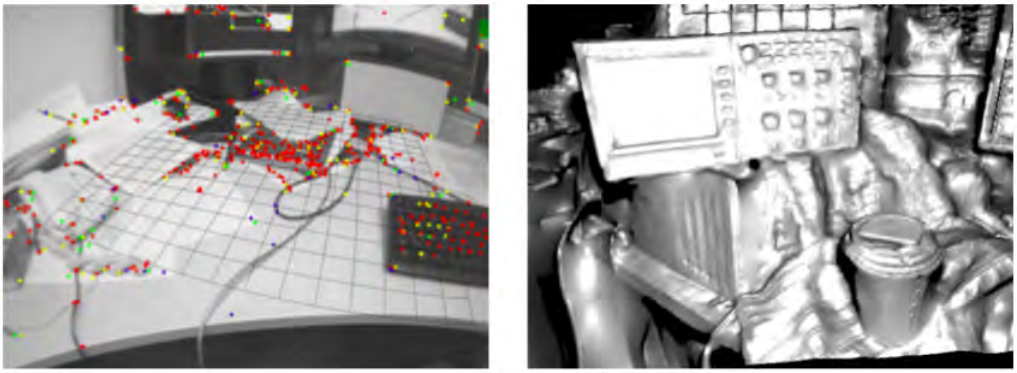

Figure 2.3: *Left:* Colored points are points of PTAM sparse map [1], *Right:* All surface points are part of the DTAM dense map [2]

Visual SLAM algorithms are classified into two categories according to how many pixels are used to reconstruct the map:

- **Sparse vs. Dense**. *Sparse* SLAM systems use only a small selected subset of the pixels in an input frame, while *dense* SLAM systems use most or all of the pixels in each received frame. Figure 2.3 shows the difference between them.

- **Feature-based vs. Direct**. *Feature-based* systems require that features are first extracted from the input images (such features may include corners, lines, curves, etc.). These features match the features obtained from different poses, and solve the SLAM problem by minimizing the feature projection error. On the other hand, *direct* systems utilize the whole image (at the pixel level) to perform localization and mapping.

2.1.2 KinectFusion

KinectFusion algorithm was the first attempt to real-time volumetric reconstruction of a scene in variable lightning conditions [21]. This reconstruction method projects each point (voxel) of the depth image within a volumetric grid which represents a model of the problem's world. Referring to the categories of subsection 2.1.1, KinectFusion is a real-time dense, direct RGB-D surface mapping and localization SLAM algorithm.

Figure 2.4 the steps to do 3D volumetric reconstruction. To be more precise, below described the functionality of each step.

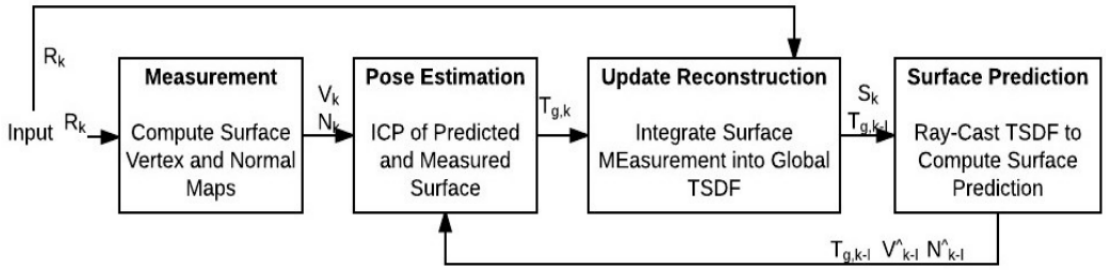

Figure 2.4: KinectFusion workflow for real-time volumetric reconstruction [3]

Surface Measurement: In this step, the depth input from the depth sensor (R_k), after applying a Gaussian bilateral filter, is used to build normal map and vertex map pyramids (N_k, V_k).

Pose Estimation: Using information gained in the previous step, an Iterate Closest Point (ICP) algorithm estimates the optimal alignment between current frame vertex and normal map pyramids and 3D world map by using steps of rotations and translations. The tracked 6 DOF camera pose is described by the rigid body transformation matrix given below (eq. 2.1), which refers to the frame g, a given time k.

$$T_{g,k} = \begin{bmatrix} R_{g,k} & t_{g,k} \\ 0 & 1 \end{bmatrix} \in \mathbb{SE}_3 \tag{2.1}$$

where the Euclidean group $\mathbb{SE}_3 := \{R, t | R \in \mathbb{SO}_3, t \in \mathbb{R}^3\}$. Any rigid transformation in 3D can be described by means of a $4x4$ matrix T with the structure above, where the $3x3$ orthogonal matrix $R \in \mathbb{SO}_3$ is the rotation matrix (the only part of **T** related to the 3D rotation) and the vector t represents the translational part of the 6D pose. For such a matrix to be applicable to 3D points, it must be represented first in homogeneous coordinates which, in our case, will consist in just considering a fourth, extra dimension to each point which will be always equal to the unity [28]. Table 2.1 derives useful information about topological groups used to describe transformations in 2D and 3D space.

Update Reconstruction: A volumetric representation (voxel grid) is defined here and fused with new information of each new input depth map. In each location (voxel)

in 3D representation of the world $p \in \mathbb{R}^3$, is assigned a signed value determined by a Truncated Signed Distance Function (TSDF) $S_k(p)$. During this phase, each voxel stores the normalized signed distance to the nearest surface, which is positive if the voxel located in-front of and negative if it is behind the surface (figure 2.5).

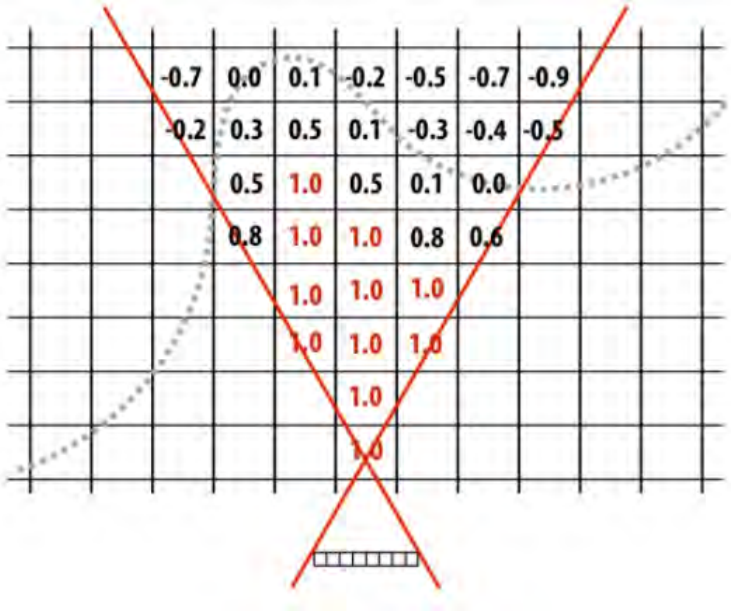

Figure 2.5: Store scene geometry as truncated signed distance function (TSDF) [4]

Surface Prediction: It is a post-volumetric integration step in which the surface is visualized by ray-casting the signed distance stored in voxel grid into an estimated frame and aligning this ray-casted view with live depth map. It constitutes a depth map (consisting of normal and vertex map) representation of the 3D world, used as described in *Pose estimation* phase.

After analysing characteristics of volume reconstruction based algorithms, which also KinectFusion complies with, the following section examine more precisely the algorithm's specific features.

KinectFusion implementation

As starting point, we use SLAMBench suite [9] which provides a KinectFusion implementation in C++, OpenMP, OpenCL and CUDA. Figure 2.6 outlines the functionality of this implementation.

The **acquisition** step reads a new RGB-D frame from the input source. This step models I/O costs during benchmarking.

Group	Description	Dim.	Matrix Representation
$\mathbb{SO}_3$	3D Rotations	3	3D rotation matrix
$\mathbb{SE}_3$	3D Rigid transformations	6	Linear transformation on homogeneous 4-vectors
$\mathbb{SO}_2$	2D Rotations	1	2D rotation matrix
$\mathbb{SE}_2$	2D Rigid transformations	3	Linear transformation on homogeneous 3-vectors
Sim_3	3D Similarity transformations (rigid motion + scale)	7	Linear transformation on homogeneous 4-vectors

Table 2.1: Lie groups frequently used to describe rigid body transformations [7]

In correspondence with the previous analysis of the algorithm, the **bilateral filter** referred to *Surface Measurement* phase. It is a stencil-based, edge-preserving filter that blurs the depth image in order to reduce the effects of noise and invalid depth values [29]. This kernel uses a 5x5 coefficients array, which combines position-based with range filtering. Range filtering averages image values with weights that decay as image dissimilarity increases. In this step, also a three-level pyramid is created by sub-sampling the filtered depth image. The generated pyramid determines the input data structure of tracking (vertex and normal map pyramid).

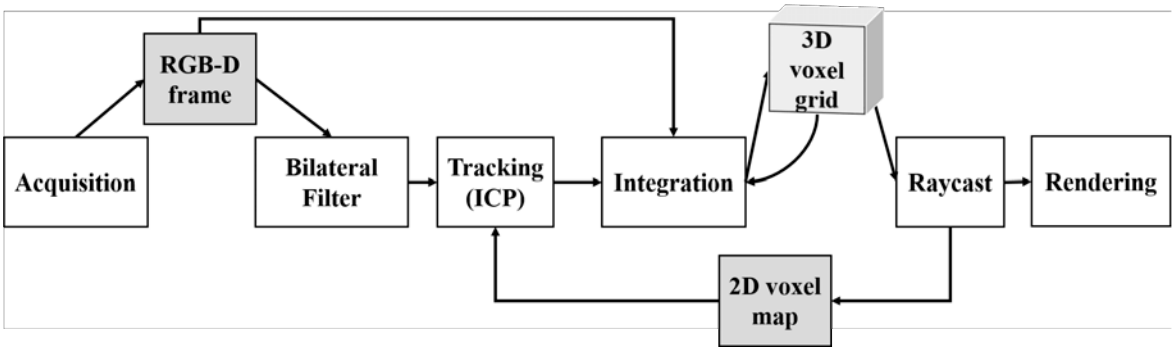

Figure 2.6: KinectFusion data processing pipeline [5].

Tracking estimates the 3D pose of the agent by registering the input depth frame with the 2D projection of the currently reconstructed model from the most recent camera position (the voxel map produced by raycasting). Tracking is based on the Iterative Closest Point (ICP) algorithm which estimates the optimal alignment between the depth and the 2D voxel map using steps of rotations and translations [30]. An additional step in *Pose estimation* phase is

reduction step which sums up all the distances of corresponding vertices of the two maps with a highly parallel tree reduction process to produce a 6×6 system of linear equations and finally compute the best match. Equation 2.1 describes the estimated camera pose. Tracking and reduction are called multiple times (until convergence or up to a maximum number of iterations) at each image of a three-level pyramid of images.

Once the new pose of the tracked frame has been determined, **integration** merges the corresponding depth map into the current 3D reconstructed model. KinectFusion 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 [31], as described above for *Update Reconstruction* phase.

The post-volumentric integration step is **ray-casting**, a computer graphics algorithm used to render 3D scenes to 2D images. From each pixel point of the final 2D image, raycast emits (casts) an imaginary ray towards the 3D voxel grid. The algorithm iteratively traverses the ray with a specific step size to select sampling points. Since these points are not typically aligned with the voxels of the grid, a trilinear interpolation step is necessary to compute the value of the sampling point from its surrounding eight voxels. Ray traversal terminates when the interpolation computes a negative value (indication that it has intercepted the surface of an object), or the ray reaches the bounding box of the 3D voxel grid (indication of empty space). Once an object is detected, a gradient of illumination values is computed that represents the orientation of the detected surface. Raycast updates the vertex and normal maps that identify the surface to the current estimate of the camera position.

Finally, **rendering** draws the constructed 2D voxel map (the output of raycasting) and the tracking trajectory of the agent. Rendering is not part of the SLAM algorithm and is not considered in our study.

2.1.3 ICL-NUIM dataset

SLAMBench introduces a alternative way of SLAM systems benchmarking. Instead of a robot and sensors live use, provide as input pre-captured video frames. There are numerous published datasets for this purpose, but we focus on ICL-NUIM [32]. The ICL-NUIM dataset aims at benchmarking RGB-D, Visual Odometry and SLAM algorithms. They present a collection of handheld RGB-D camera sequences within two different synthetically generated environments (the living room and the office room) with ground truth information.

Figure 2.7 shows how the baseline KinectFusions algorithm manipulates the input depth scene. The input RGB and Depth images describe a frame of the third living room trajectory (lt_kt2) of ICL-NUIM dataset.

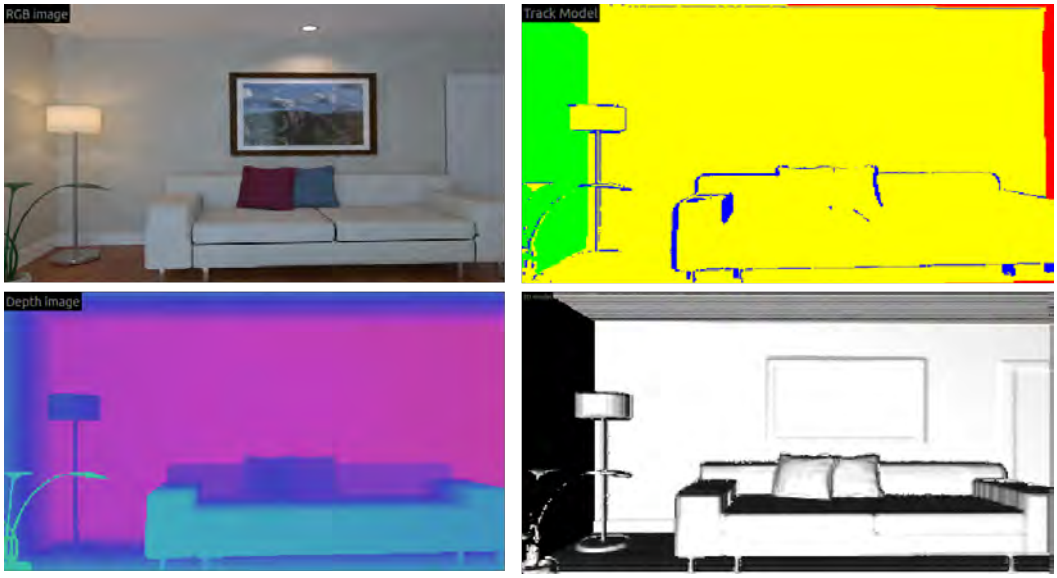

Figure 2.7: KinectFusion I/O: The input RGB scene (top left), the input depth frame (bot. left), the tracking output (top right), and the reconstructed 3D map (bot. right).

2.1.4 Error evaluation

In general, a SLAM system produces the approximate agent trajectory. The quality of the approximate trajectory from a given input sequence of RGB-D images, it is necessary to be measured. Both sequences consist of homogeneous transformation matrices that express the pose of the RGB-D frame of the Kinect from an (arbitrary) reference frame.

For visual SLAM systems, the **Absolute Trajectory Error (ATE)** is frequently used as a trajectory error metric. ATE accumulates the error across all frames of a trajectory and can be evaluated by comparing the absolute distances between the predicted trajectory and the ground truth. As both trajectories can be specified in arbitrary coordinate frames, they first need to be aligned, by using the method of Horn [33], which determines the rigid-body transformation corresponding to the least-squares solution that maps the estimated trajectory into the ground truth trajectory. ATE is computed as the sum of **Root Mean Square Error (RMSE)** between the aligned ground truth and estimated trajectory of all frames. Since ATE depends on the number of frames in the trajectory, in this work we use the average RMSE across all frames, as the error metric.

2.2 FPGA Technology

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a uniform matrix of configurable logic blocks (CLBs) that are interconnected by a configurable routing grid. CLBs are digital circuits which form the core of the FPGA's programmable-logic capabilities. The CLBs interact with each other and with external circuitry. For these purposes, the FPGA uses a matrix of programmable interconnects and input/output (I/O) blocks. CLBs include look-up tables (LUT), storage elements (flip-flops or registers), and multiplexers that allow the CLB to perform data-storage, as well as logic and arithmetic operations.

Programmable routing grid [34] consists of interconnecting wires that join CLBs together to build complex logic. It also joins I/O blocks to CLBs and joins CLBs to memory resources as figure 2.8 shows. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.

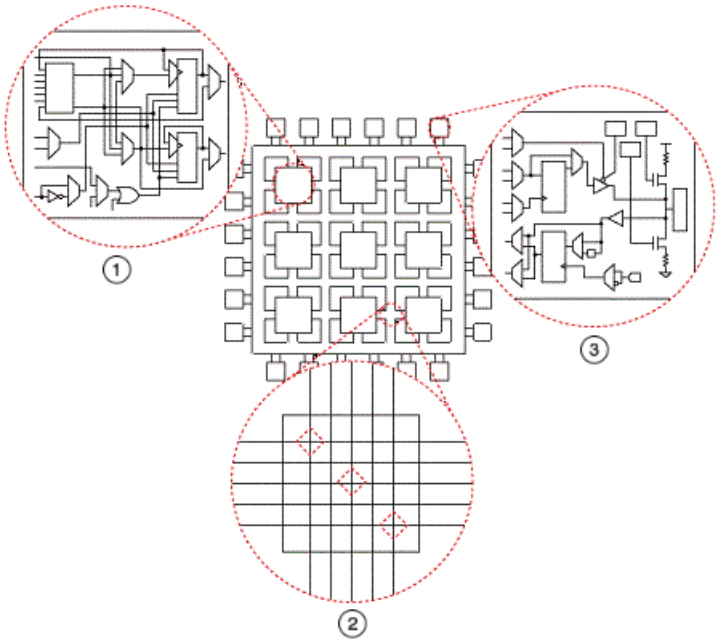

Figure 2.8: FPGA structure: 1. Configurable logic blocks (CLBs), 2. Programmable routing, 3. I/O Blocks

For an application execution, there is a need for an FPGA-based design which defines the required computing tasks to transform this collection of thousands of hardware blocks into a correct configuration. FPGA programming achieved either by traditionally using a hardware

description language (HDL) such as VHDL or Verilog or through software development kits (SDKs) and High-level synthesis (HLS) design tools that allow designers to develop FPGA solutions in popular high-level languages such as C/C++, Python and OpenCL.

FPGAs provide significant advantages compared with fixed-hardware platforms (CPUs, GPUs), such as low latency/high performance and high power/energy efficiency. Due to their programmable nature, FPGAs are an ideal fit for many different markets and applications such as Aerospace & Defense, ASIC Prototyping, Audio, Automotive, Consumer Electronics, Data Center, High Performance Computing and Data Storage, Industrial, Medical, Security, Video & Image Processing, Wired and Wireless Communications. In general, FPGA performance strongly depends on the application characteristics.

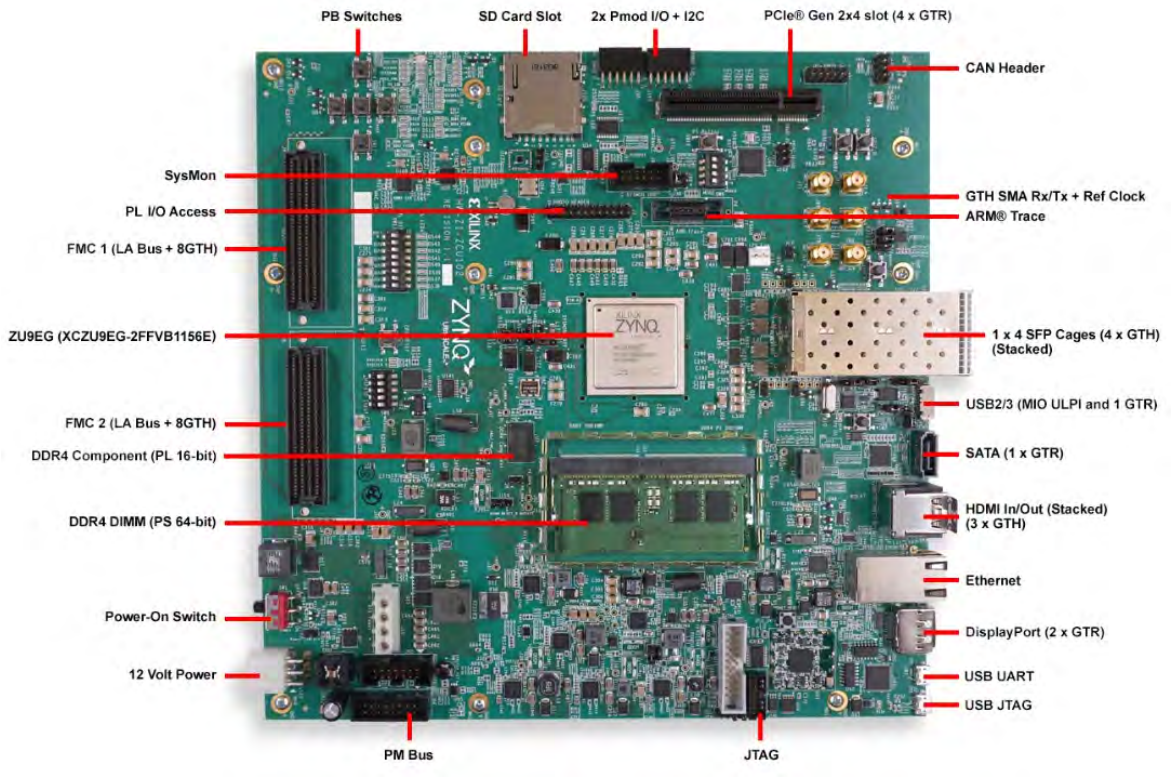

Figure 2.9: Zynq UltraScale+ MPSoC ZCU102 [6]

Multiprocessor System on a Chip (MPSoC) FPGA

MPSoC FPGA devices integrate both processor and FPGA architectures into a single device. This provides higher integration, lower power, smaller board size, and high-bandwidth communication between the processor and FPGA. MPSoC also include a set of peripherals, on-chip memory, an FPGA-style logic array, and high speed transceivers. In this thesis,

we use the Ultrascale+ ZCU102 MPSoC (Multi-Processor SoC) board as a target platform (Figure 2.9).

2.3 Related work

It is only recently that dense SLAM algorithms have been ported in an FPGA. In [35], Gautier at al. perform an extensive design space exploration is performed for the InfiniTAM algorithm [36] (which is itself derived from KinectFusion [21]). They develop by developing a large number of parameterized architectures for each of the dense SLAM components. The performance of the different OpenCL-based InfiniTAM implementations is evaluated using a low-cost Terasic DE1 FPGA System-on-Chip (SoC), and a high-performance Terasic DE5 PCIe board, achieving less than 2 fps and 44 fps on an 320x240 input depth image, respectively. Most of the improvements wrt. to the SW-only implementation are due to the merging of the depth fusion and raycasting kernels, loop-based optimizations and caching/prefetching strategies. Unlike our work, the authors only focus on performance and do not explore any accuracy vs. performance trade-offs. Since KinectFusion is a closed-loop algorithm, some of the excessive approximations described in [35] may result in very large ATE. Earlier work by the same authors only maps the tracking and depth fusion kernels in the FPGA [37].

ORB-SLAM is a feature-based sparse SLAM system that operates in real time without acceleration in indoor and outdoor environments [16]. A hardware implementation of ORB-SLAM is presented in [38]. Abouzahir et al. [39] evaluate a number of sparse SLAM algorithms in desktop and embedded platforms and also implement FastSLAM 2.0 [40] in a GPU and an Arria 10 FPGA.

To bridge the gap between dense and sparse SLAM approaches, a family of semi-dense SLAM algorithms such as LSD-SLAM [15] have been proposed to provide a more dense and information-rich representation compared to sparse methods, while achieving better computational efficiency from processing a subset of high quality observations. In [41], the LSD-SLAM algorithm is accelerated on an FPGA SoC, achieving more than 60 mapped fps on a 640x480 input visual frame. Previous work by the same authors describes an FPGA accelerator only for semi-dense tracking (but not mapping) on embedded platforms [42]. A very low-power ASIC design for real-time visual inertial odometry (VIO) targeting nano-drones has been announced recently [43]. The chip, fabricated at 65nm CMOS technology, uses in-

ertial 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.

Approximate computing has been used to accelerate SLAM implementations. [44] studies the performance impact of reduced-precision floating-point arithmetic in SLAM algorithms. In SLAMBooster, the degree of approximation is dynamically adjusted during the motion of the robot [45]. 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.

Chapter 3

Precise and Approximate Optimizations on KinectFusion

In this chapter, we describe our proposed MPSoC FPGA-based architectures along with the most significant performance optimizations for each kernel described in section 2.1.2. The main objective is to maximize the throughput for each individual kernel and the complete algorithm without considerably increasing the localization error.

Code optimizations are classified in the following two categories:

- **precise** optimizations, which retain the accuracy of the baseline code, and
- **approximate** optimizations, which may affect the accuracy of the baseline code

As we already mentioned in chapter 1, our starting point is the C++/OpenMP SLAMBench suite implementation of KinectFusion algorithm. The first step is to transform the C++ baseline code to FPGA-oriented C++ code by applying basic hardware optimizations. These precise optimizations for all kernels include the usual assortment of loop unrolling and software pipelining (with various factors and iteration intervals II, respectively) and using prefetching and BRAM array partitioning so that input elements are only read once from external DRAM.

For each kernel, we apply the approximate optimizations on the fastest precise implementations, i.e. after the repertoire of precise optimizations has been exhausted. To increase throughput, we aim at (i) reducing execution time of each kernel by selectively combining both precise and approximate optimizations, and (ii) scheduling multiple accelerators for parallel execution.

Bilateral Filter		**Tracking**		**Integration**		**Raycast (SW)**	
Optim.	Description	Optim.	Description	Optim.	Description	Optim.	Description
BF_Pad	Pad input frame	**Tr_Unroll**	Unroll inner loop & cache in BRAM	**Int_Inter**	Loop Interchange		
BF_Unroll	Unroll inner loop & cache in BRAM	**Tr_Pipe**	Pipeline inner loop to process a pixel after II=1 cycles	**Int_Unroll**	Unroll inner loop & cache in BRAM		
BF_Pipe	Pipeline inner loop to process a pixel after II=1 cycles	**Tr_NCU**	N Compute Units	**Int_Pipe**	Pipeline inner loop to process a voxel after II=1 cycles		
BF_NCU	N Compute Units			**Int_NCU**	N Compute Units		
BF_Coeff	3x3 coefficients	**Tr_LP**	Loop perforation	**Int_SLP**	Loop perfor. (Skip)	**R_Step**	Larger ray steps
BF_HP	fp16 arithmetic	**Tr_HP**	fp16 arithmetic	**Int_CLP**	Loop perfor. (Copy)	**R_LP**	Skip computing of rays
BF_Range	No range filter	**Tr_LvlIter**	Skip pyramid levels & reduce max # iter.	**Int_HP**	fp16 arithmetic	**R_TrInt**	Use fewer points for trilinear interp.
				Int_Br	Eliminate checking		
				Int_FPOp	Eliminate expensive FP Operations	**R_Fast**	Use –ffast-math
						R_Rate	Skip one frame

Table 3.1: Precise (top rows) and Approximate (bottom rows) optimizations for the KinectFusion kernels.

The following sections contain an analysis of each kernel precise and approximate optimizations shown in table 3.1. Some of the optimizations in the table are widely used as techniques when accelerating an application on FPGAs. These are:

- multiple instances of the same accelerator (*NCU*), used to process different parts of the input frame concurrently to further increase performance.
- use of half precision floating-point arithmetic or fp16 (*HP*), which mainly reduces the FPGA utilization by replacing single precision (32-bit) FP operations with 16-bit arithmetic. This leads to an increase in error of FP operations due to the use of fewer

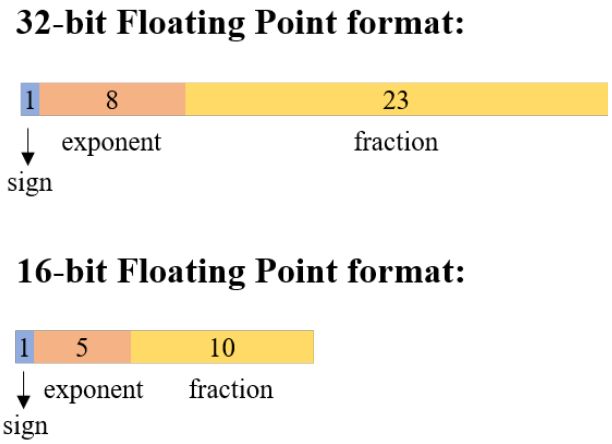

Figure 3.1: Floating Point formats according to IEEE 754 standard.

bits to describe FP numbers as shown in figure 3.1.

3.1 Bilateral Filter

The bilateral filter (BF) kernel reads the input depth image which is a 2D array of $320x240$ elements, the $5x5$ convolution filter and writes back to an $320x240$ image.

Precise optimizations:

(i) The input depth frame is padded (BF_Pad) by the host CPU with additional rows and columns to eliminate checking of boundary conditions, which is a major limitation for pipeline parallelism in FPGAs.

(ii) Unrolling the inner loop (BF_Unroll) which computes each input pixel convolution has a significant performance impact.

(iii) Loop pipelining (BF_Pipe) is used to overlap loop iterations. Due to the need of multiple read accesses of the same buffer positions, for each output image line, five rows of input structure are stored in BRAM applying the corresponding array partitioning.

(iv) Multiple instances of the kernel (BF_NCU) parallel the execution of independent parts of the image.

Approximate optimizations: Bilateral filter kernel approximate optimizations mainly affect FPGA utilization, without significant increase in error.

(i) We use a smaller 3x3 coefficient array (BF_Coeff) instead of the nominal 5x5 array. Although a convolution filter with smaller radius results to a less smooth image, this replacement does not affect the system's accuracy for the specific input trajectory. Moreover, this optimization reduces FPGA utilization and results into fewer BRAM memory accesses, which leads to higher performance.

(ii) We use half precision 16-bit floating point format (fp16) instead of 32-bit float (BF_HP).

(iii) We remove the range filtering stage of the bilateral filter (BF_Range), which eliminates the invocation of an exponent function in the coefficient filter.

3.2 Tracking

The tracking kernel accesses the current depth image row-wise and pairs each input pixel to its corresponding point in the 2D projection of the reconstructed model. Therefore, some accesses follow row-wise direction and other read voxel grid 2D representation randomly depending on the camera pose. Tracking is invoked multiple times for the three pyramid levels of each frame. In particular, the default maximum number of iterations is 10, 5, and 4 for the level 0 (the higher-resolution frame) and for levels 1 and 2, respectively.

Precise optimizations: We apply software pipelining and loop unrolling (Tr_Pipe and Tr_Unroll) coupled with the provision to cache input and/or output data rows in BRAMs. Moreover, we evaluate the option to merge the tracking and reduce kernels into a larger kernel. Additionally, we use multiple instances of tracking accelerator (Tr_NCU) to process concurrently different parts of the image.

Approximate optimizations:

(i) Loop perforation is an approximate computing technique used to skip loop iterations [46]. We employ this technique to skip pixels of the input depth frame (Tr_LP). For each 2D part of input with dimensions 4x8, the updated top left pixel value is assigned to the rest pixels of the block.

(ii) We replace all FP operations with half precision fp16 arithmetic (Tr_HP).

(iii) We reduce pyramid levels and skip tracking kernel invocations for each one of them (Tr_LvlIter). Pyramids are frequently used for image detail enhancement and manipulation. However, the smaller the image dimensions the fewer the details of an image. In our application, the depth resolution is 320x240 which implies that skipping processing of lower resolution images will not substantially degrade image quality.

3.3 Integration

This kernel consists of a triple nested loop which updates the 3D voxel grid (consisting of 256x256x256 voxels) using the new pose of the agent obtained by tracking. Each voxel can be processed independently, which, combined with the row-wise access of the 3D voxel grid suggests that the Integration kernel has a high acceleration potential in the FPGA.

Precise optimizations: Besides the usual assortment of loop optimizations, we also perform loop interchange (Int_Inter) to improve the memory access patterns of the 3D grid and thus increase the overall code spatial locality. Without the achieved spatial locality, any effort according to the loop pipelining and optimal memory access pattern does not pay off. Then, we use internal BRAMs to prefetch and cache a variable number of TSDF rows to overlap computing with memory reads. Thirdly, we apply software pipelining (Int_Pipe) and unrolling (Int_Unroll) and allocate data in internal BRAMs to maximize bandwidth. Finally, we instantiate multiple instances of the kernel to partition the 3D voxel processing into N parts (Int_NCU).

Approximate optimizations:

(i) Loop perforation is used to skip iterations of the triple nested loop For each frame, the TSDF values of the skipped iterations are evaluated as a function of TSDF values that have been computed conventionally. We experimented with a variety of approaches which include (a) skipping altogether the computation of new TSDF values (Int_SLP), and use, instead, the old TSDF values (i.e the values at the same position in the previous frame), (b) using the average or just copying the newly computed TSDF values of neighboring positions to the skipped positions (Int_CLP). Experimental evaluation indicates that the maximum number of skipped iterations which does not affect system accuracy is six (along the z axis).

(ii) We use the half-precision (fp16) format instead of high precision floating-point operations (Int_HP).

(iii) We eliminate some of the branches that are used in the loops to check for special conditions (Int_Br). However, most of them constitute a significant role in the algorithm's flow.

(iv) We replace expensive floating point operations (Int_FPOp) which have low variation across loop iterations with simpler functions or constant values. For example, we remove the calculation of a square-root operation used as a normalized factor in the distance between the current pixel and the estimated camera position in the z-axis.

3.4 Raycasting

Even though all rays can be traversed concurrently as separate threads (thus facilitating parallel execution by multiple accelerators), accessing the TSDF values results in a non-contiguous and irregular memory access pattern, making memory prefetching, data distribution to internal BRAMs, and data reuse very challenging. For these reasons, the raycast kernel is executed on the ARM CPU in all implementations.

Approximate optimizations:

(i) Ray traversal uses steps of variable size (R_Step). The step starts with a larger size, which becomes smaller as the ray approaches a surface or the edges of the 3D voxel grid. We use steps of constant size to achieve a deterministic schedule and size of voxel prefetching.

(ii) We only compute along a fraction of rays (similar to loop perforation). For a 2x2 block of neighboring rays, when one of them (top left) reaches an object's surface, then we assume the same for the rest of them (R_LP).

(iii) Raycast trilinear interpolation accesses 8 different TSDF values for averaging. We provide the option to use simpler interpolation schemes that require only two TSDF accesses (R_TrInt).

(iv) We use fast math (R_Fast) to approximate expensive floating point operations.

(v) We perform raycast at a lower frequency, once every two frames (R_Rate).

Chapter 4

Experimental evaluation

This chapter presents the effect and significance of each optimization described in chapter 3. It also presents the fastest configurations performance, power and area analysis results.

4.1 Methodology

We implemented our designs using the Vitis™ Platform targeting the Xilinx UltraScale+ MPSoC ZCU102 Evaluation Kit. The 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 runs at 300 MHz in all our experiments. For comparison, we use the SW-only KinectFusion implementation on a desktop platform with a 3.6 GHz Intel Xeon® W-2123 processor with 4 cores (8 hardware threads) and 16 GB DDR4 main memory. Figure 4.1 shows the contribution (%) of each kernel to the total execution time, when running the C++/OpenMP KinectFusion implementations on an ARM Cortex-A53 and a Xeon W-2123 CPU, as well as the application throughput without including the rendering and acquisition kernels. Note that the integration and raycast kernels contribute more than 70% of total execution time.

To compare power consumption between the two architectures (FPGA, x86), we measure power dissipation on the FPGA using the Power Management Bus (PMBus) protocol which monitors multiple power rails, and on Xeon via the MSR power monitoring registers. [1]

As input, we use four camera trajectories lr.kt[0-3] from ICL-NUIM dataset (described in section 2.1.3), a synthetic dataset providing RGB-D sequences from a living room model [32].

[1] Power estimation was done by Alexandros Patras.

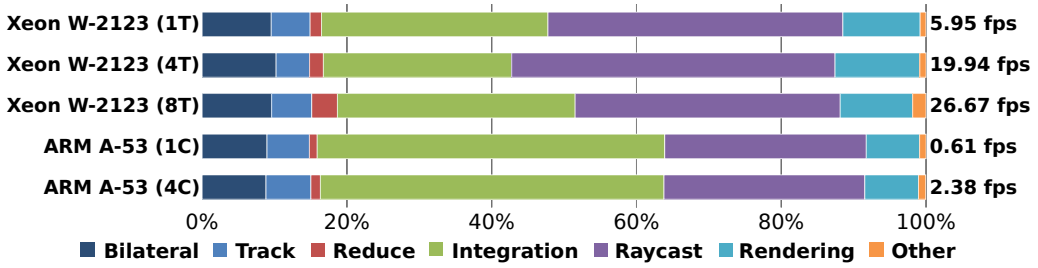

Figure 4.1: Contribution of each KinectFusion kernel to total execution time (T=#threads, C=#cores). The y-axis to the right shows throughput in frames/sec.

We run all frames of each trajectory, lr.kt0:1510, lr.kt1:967, lr.kt2:882, lr.kt3:1242, but, for the evaluation of our optimizations we mainly use lr.kt2. Note that precise optimizations have the same performance effect for all trajectories, but approximate optimizations may be trajectory dependent. In all experiments, we do not consider rendering as part of the KinectFusion pipeline. We use the default SLAMBench parameters as follows:

- Input depth frame resolution is 320x240.
- Integration and tracking rate is once per frame.
- Number of pyramid levels in tracking is 3.
- Maximum number of tracking iterations is 10,5,4 for levels 0, 1 and 2, respectively.
- Volume resolution for the scene reconstruction is 256^3 voxels.
- Volume map (dimensions of the reconstructed scene) is $4.8^3 m^3$.

The output error metric is the average (across all frames) root mean square error (RMSE) between the ground truth and the estimated trajectories of the agent as described in section 2.1.4. Figure 4.2 is the visualization of three lr.kt2 trajectories:

(i) the ground truth trajectory (RMSE=0),

(ii) the baseline KinectFusion trajectory (RMSE=2cm), and

(iii) a trajectory with RMSE=3.4cm.

Note that even the baseline KinectFusion algorithm is approximate with an average RMSE equal to 2cm. Following the same procedure for a configuration with RMSE equals to 3.4cm,

we do not observe too large error deviation. Moreover, a frame is defined as *untracked* when the algorithm fails to recognize its position in the map. We have found experimentally that an average RMSE higher than 3.4cm (for lr.kt2) almost always results into more untracked frames.

Figure 4.2: Trajectories with various average RMSE values. Grid unit distances are 0.5m in x-axis, 0.05m in y-axis, and 0.5m in z-axis.

4.2 Design Space Exploration

As already described in chapter 3, the optimizations we use are categorized into precise and approximate. Precise optimizations such as software pipelining, loop unrolling, BRAM partitioning are used to improve performance without negatively affecting the RMSE of the trajectory. Concerning approximate optimizations, we place an upper bound on the average RSME which is trajectory-specific but does not exceed 3.4cm for the *lr.kt2* trajectory. Figure 4.3 shows the execution time of each kernel (N=1) when implemented in hardware (using Vitis HLS). Table 4.1 shows the throughput (in Hz) and the speedup achieved by the fastest precise and approximate hardware implementations compared with the unoptimized hardware implementation. The last column shows the average RMSE per frame when only the corresponding kernel is approximate and all other kernels run precisely.

For a single *bilateral filter* HW accelerator, precise optimizations yield $705x$ speedup compared with the unoptimized HW implementation, whereas approximate optimizations further increase the speedup to $1044x$ (Table 4.1). For the *integration* kernel, the speedups are $14x$ and $59x$, respectively. These two kernels are characterized by regular memory access

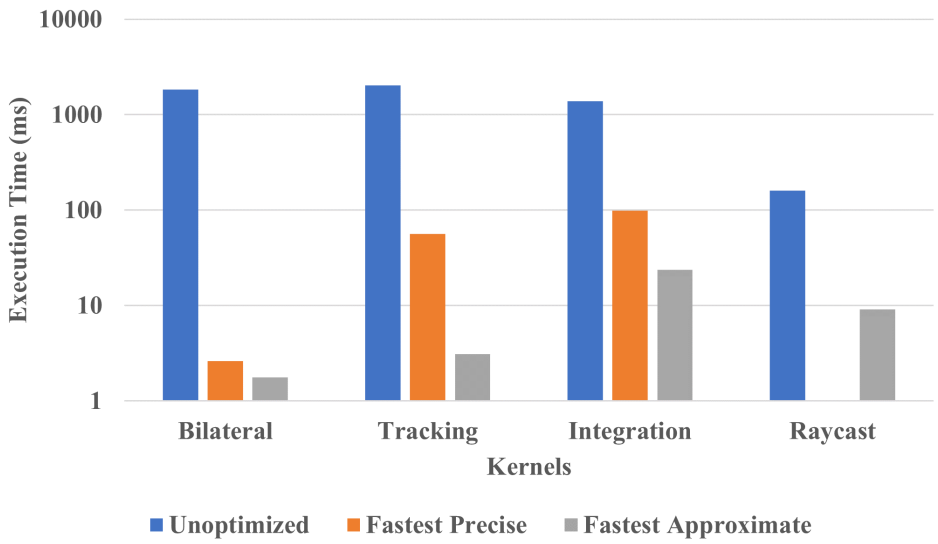

Figure 4.3: Execution time of each kernel for the Unoptimized, Fastest Precise and Approximate versions. Note the logarithmic scale of the y-axis .

	Unopt.	**Fastest and Precise**	**Fastest and Approximate**	
	Hz	Hz (Speedup)	Hz (Speedup)	RMSE
Bilateral	0.54	380.5 (705x)	564 (1044x)	2.28
Tracking	0.49	17.7 (36x)	323 (659x)	2.02
Integration	0.72	10.1 (14x)	42.4 (59x)	2.54
Raycast (SW)	6.28	-	110 (17.5x)	2.07

Table 4.1: Performance of HW and SW kernel implementations (1 accelerator).

patterns (2D and 3D, respectively), and readily available coarse-grain parallelism. Interestingly, loop perforation (*Int_SLP*) in the vertical z-axis also provides substantial speedup of up to 5x for 5 skipped iterations, without noticeable loss of accuracy. This is because the 3D voxel grid is not updated frequently along this direction.

For the *tracking* kernel (the slowest unoptimized kernel at 0.49 Hz), the execution time decreased from 2026ms (unoptimized) to 3ms (approximate) at a 59x speedup.

As we mentioned in section 3.4, the *raycast* kernel is not suitable for hardware acceleration mainly due to complex memory accesses in the large 64 MB 3D voxel grid. Given this, in table 4.1 and figure 4.3 the corresponding results referred to the software implementation of the kernel, and yielded 25.9x speedup.

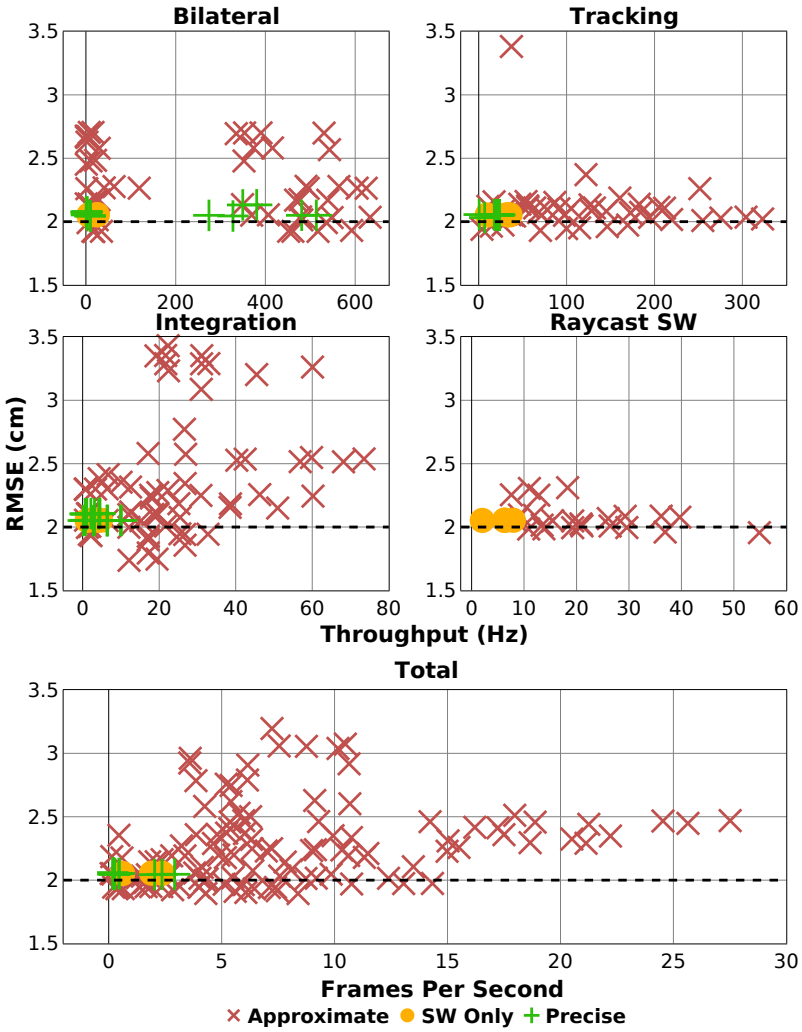

Figure 4.4: The scatter plots show throughput vs. average RMSE of various SW and HW kernel implementations when running the *lr:kt2* benchmark.

Figure 4.4 shows the results of the throughput vs. average RMSE design exploration of precise and approximate optimizations for each kernel separately and for the entire application. Each red mark corresponds to an approximate HW configuration using a combination of the precise and approximate optimizations shown in Table 3.1. In *Total*, raycast is executed in the ARM CPU and all other kernels in HW. Precise HW in the *Total* plot corresponds to a configuration in which all kernels are executed precisely and each one has a single instance. Using reduced fp16 precision has proven beneficial for the hardware implementation of most kernels, primarily due to the reduced FPGA resources and the potential for fitting multiple accelerators in the fabric. In this way, approximate total configurations include multiple instances of some kernels. We found that *Int_NCU* provides speedup almost linear to the number of accelerators N and is more beneficial than Tr_NCU and R_NCU because the

integration kernel accesses a bigger data structure (3D) compared with bilateral and tracking. As it is validated in the following section 4.3, without Int_NCU enabled the execution time is too high to achieve real-time execution.

4.3 Significance Analysis

To better quantify the significance of individual optimizations of Table 3.1 on performance and RMSE, and provide a systematic approach to select optimizations for a given performance target, we use Lasso [47], a regularized linear regression method that builds models to fit objective functions (e.g. throughput, RMSE). Lasso tends to favor the case when there is a small number of significant optimizations by pushing the coefficients of insignificant optimizations to zero. The result of this analysis are coefficients θ_j for each feature j, which represent the relative contribution of j to performance, aiming at minimizing the following loss function:

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

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

We perform Lasso analysis for each kernel separately and for the whole application. Coefficients with a high absolute value indicate a stronger correlation between the corresponding feature and performance. Since using features with higher degree provides better model accuracy (lower mean square error, MSE), we choose to report features up to the second degree for each kernel. Using only first degree features typically results in a higher MSE which indicates non-linear relations between optimizations and performance.

Table 4.2 shows the features with the largest coefficients for *bilateral filter* throughput

Table 4.2: Lasso analysis of KinectFusion Bilateral Filter

Throughput (MSE=0.039)		**RMSE** (MSE=0.0019)	
Feature	Coeff.	Feature	Coeff.
BF_Unroll^2	0.126	BF_Coeff^2	0.408
BF_Pipe*BF_Unroll	0.099	BF_Range	-0.237
BF_Pipe	-0.095	BF_Coeff*BF_Range	0.056
BF_Unroll*BF_Coeff	0.064	BF_Coeff*BF_HP	-0.048
BF_Coeff	-0.060	BF_HP*BF_Range	0.023

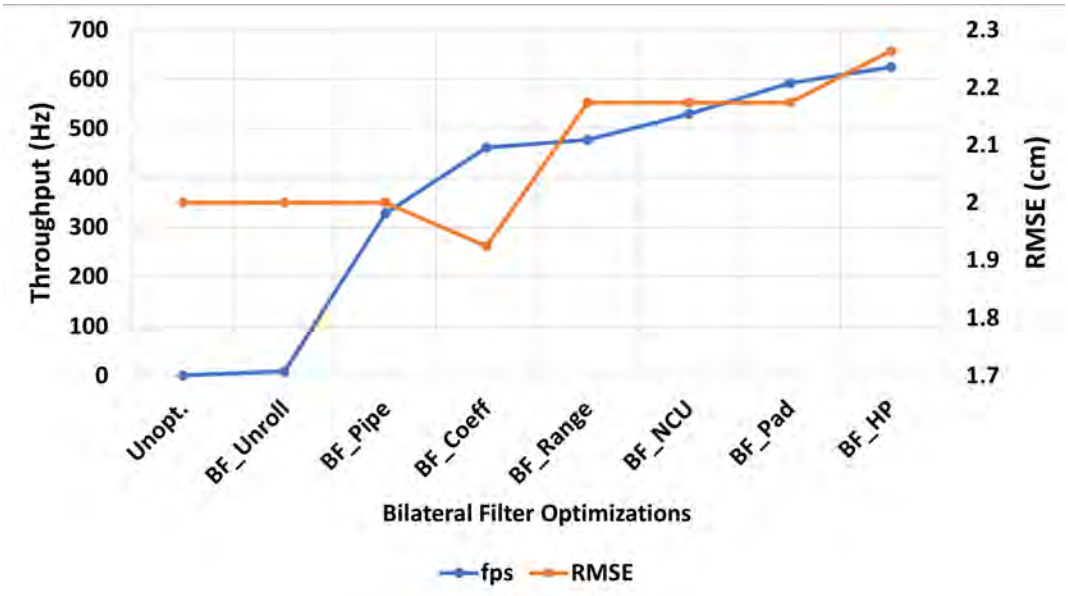

Figure 4.5: Lasso analysis validation for the Bilateral Filter kernel.

and average RMSE. Figure 4.5 is used as validation of this ranking. The x-axis consists of a sequence of optimizations sorted by the largest Lasso coefficient. Each x value means that the current optimization and all optimizations to the left are enabled. RMSE analysis shows that enabling 3x3 convolution filter, the RMSE decreases (positive Lasso coefficient, also described in section 3.1), whereas BF_Range and BF_HP have a negative impact on localization error.

Figure 4.6 shows the Lasso ranking of *tracking* optimizations of Table 4.3. In particular, the first three of them almost yield the final kernel throughput. Concerning approximate optimizations, Lasso analysis validates that both Tr_LP and Tr_LvlIter increase error.

Figure 4.7 shows that *integration* need 5 enabled optimizations (more than the previous

Table 4.3: Lasso analysis of KinectFusion Tracking

Throughput (MSE=0.077)		**RMSE** (MSE=0.008)	
Feature	Coeff.	Feature	Coeff.
Tr_Pipe^2	0.078	Tr_LP	-0.091
Tr_Pipe*Tr_LP	0.062	Tr_LvlIter	-0.060
Tr_LP	-0.039	Tr_LvlIter*Tr_NCU	0.046
Tr_Pipe*Tr_LvlIter	0.031	Tr_LvlIter^2	0.043
Tr_LvlIter	0.021	Tr_LP^2	0.024
Tr_Pipe	-0.003	Tr_Pipe * Tr_LP	-0.010

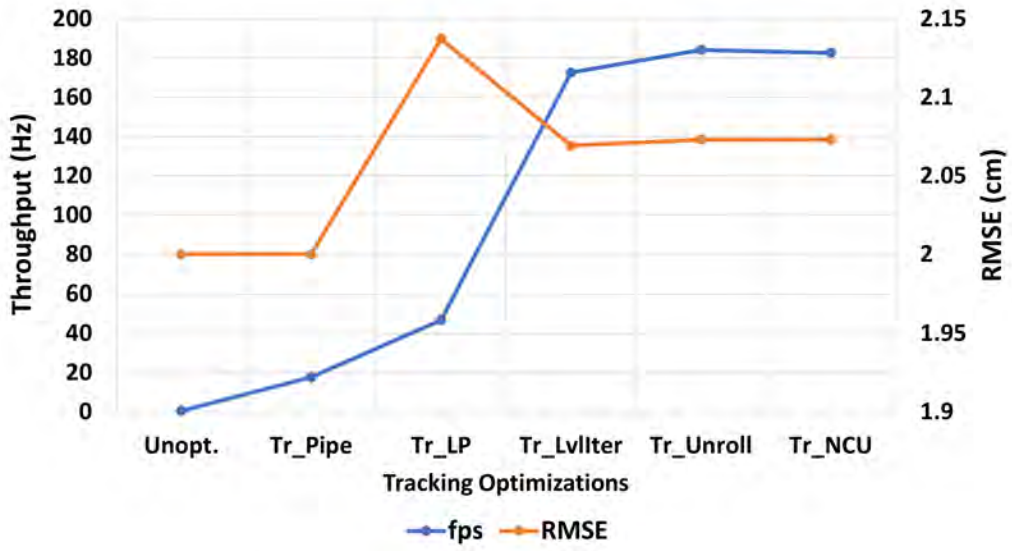

Figure 4.6: Lasso analysis validation for the Tracking kernel.

kernels need) to reach its best performance due to each optimization contribution to the time reduction is small. We also observe that Int_NCU belongs to this set, but, as mentioned in section 4.2, to achieve this performance for the entire application, approximate optimizations which reduce FPGA resources must be enabled, due to area constraints. In table 4.4, the column of throughput determines the order in which integration optimizations appears in figure's 4.7 x-axis, and RMSE analysis confirms that all the applied approximations lead to error increment.

Raycast kernel is a special case since each new optimization that is activated has the same speedup (Fig. 4.8). Concerning error analysis, all the applied approximate optimizations have a negative effect on application's RMSE, but the combination some of them, such as R_TrInt

Table 4.4: Lasso analysis validation for the Integration kernel.

Throughput (MSE=0.0008)		**RMSE** (MSE=0.017)	
Feature	Coeff.	Feature	Coeff.
Int_Pipe^2	0.184	Int_SLP*Int_CLP	-0.255
Int_Pipe*Int_Inter	-0.167	Int_CLP^2	-0.219
Int_Inter^2	0.156	Int_FPOp^2	0.169
Int_SLP	-0.120	Int_SLP	-0.083
Int_Inter*Int_SLP	0.113	Int_HP	-0.072
Int_Unroll	-0.039	Int_SLP^2	-0.06

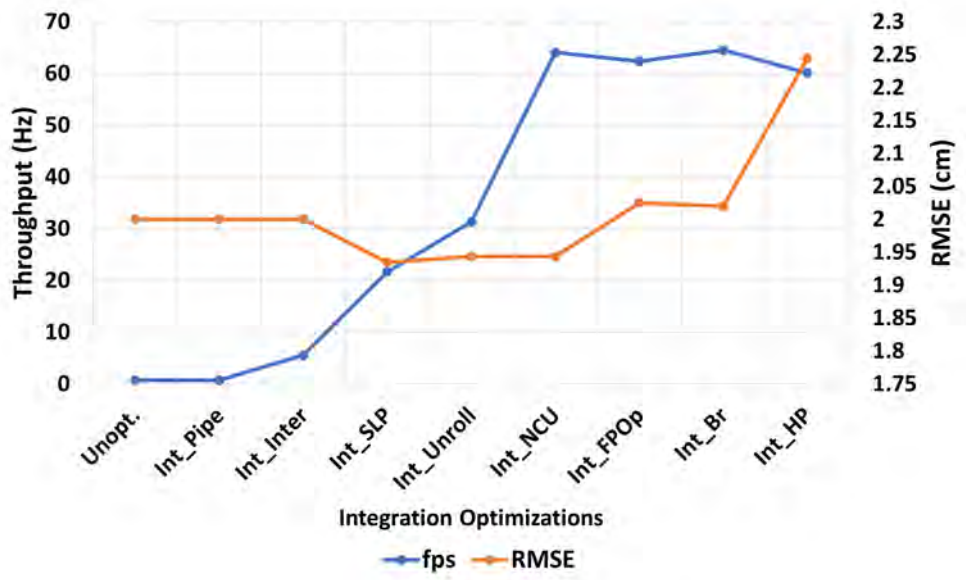

Figure 4.7: Lasso analysis validation for the Integration kernel.

and R_Step, keeps the error almost unchanged.

The Table 4.6 shows the coefficients of the most impactful features of the full KinectFusion design. As expected, the dominant optimizations are related to inner loop implementation such as loop interchange, pipelining, unrolling and caching of pixels/voxels to internal BRAMs. Approximate optimizations such as loop perforation and half precision arithmetic also score high. However, the combined analysis could not be as detailed as of the individual ones for each kernel, because too many configurations are required (27 optimizations-features generate 2^{27} runs-samples). For this reason, the validation table 4.6 reports grouped optimizations impact on performance.

The second column of Table 4.6 presents a similar Lasso analysis to quantify the effects on

Table 4.5: Lasso analysis of KinectFusion Raycast SW

Throughput (MSE=0.0016)		**RMSE** (MSE=0.038)	
Feature	Coeff.	Feature	Coeff.
R_LP	-0.145	R_LP^2	-0.166
R_Rate	-0.110	R_TrInt^2	0.166
R_Step	-0.057	R_LP	0.127
R_Fast	-0.056	R_LP*R_TrInt	0.096
R_LP*R_Rate	0.049	R_TrInt	-0.084
R_TrInt	-0.047	R_Step*R_TrInt	0.072

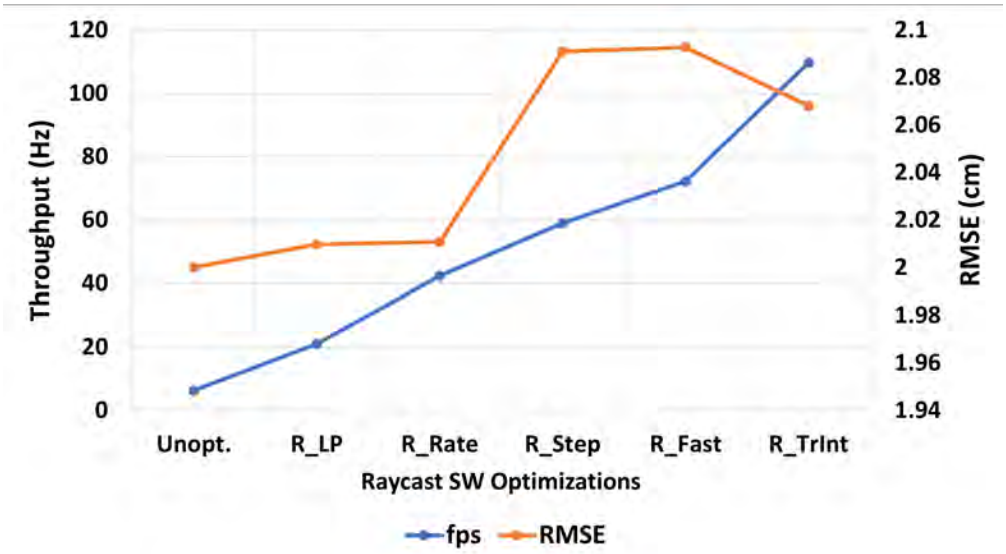

Figure 4.8: Lasso analysis validation for the Raycast kernel (SW).

the average RMSE due to the approximations applied on all kernels in combination. Typically, loop perforation, if not applied judiciously, has the most negative effect on output error.

Table 4.7, shows how Lasso ranking is used to cumulatively apply optimizations according to their impact on throughput. The *Baseline* configuration executes KinectFusion at 0.18 fps using one unoptimized accelerator for each kernel (and raycast in SW). *Conf1* uses only the 5 most impactful optimizations according to Lasso ranking in the *Throughput* column of Table 4.6 to achieve 11.2x speedup. Then, *Conf2* incrementally includes the 15 optimizations shown for each individual kernel in the corresponding Lasso analysis tables, which are included in the qualitative total analysis When the first 22 (out of 27) optimizations of the Lasso ranking are enabled, the entire application achieves top performance at 27.5 fps. The

Table 4.6: Lasso analysis of KinectFusion Combined kernels

Throughput (MSE=0.0016)		**RMSE** (MSE=0.038)	
Feature	Coeff.	Feature	Coeff.
Int_Pipe^2	0.085	R_Step	-0.035
Int_Inter*Int_Pipe	-0.056	Int_SLP	0.032
Tr_Pipe^2	0.044	Tr_LvlIter*R_LP	-0.027
Int_Pipe*Int_Unroll	-0.043	Int_SLP*Tr_LvlIter	-0.025
Int_Inter*Int_Unroll	0.039	Tr_LP	-0.021
BF_Unroll^2	0.037	Tr_LvlIter*R_Interp	0.017

Table 4.7: Optimization Selection Based on Lasso ranking.

	Baseline	**Conf1**	**Conf2**	**All Optimiz.**
Throughput (fps)	0.18	2	15.6	27.5
Speedup wrt. Baseline	1	11.2	86.2	151.7
Speedup wrt. previous configuration	-	11.2	7.69	1.76

last 5 optimizations resulted into implementations that exceeded available FPGA resources and their Lasso coefficients are zero. As this statistical analysis indicates, additional optimizations after *Conf1* are still important to increase performance further although at diminishing returns.

Besides inter-frame, kernel-based optimizations, we also tried to overlap execution of multiple frames to increase accelerator utilization and further increase performance. Due to frame recurrent computations (back edge in Fig. 2.6), we can only overlap the bilateral filter, gaining 4 ms per frame.

The fastest approximate configuration consists of one bilateral, one tracking and four integration kernels (all approximate), and has average RMSE equal to 2.5cm. Approximate computing not only enables faster individual kernel executions, but it also makes a better use of FPGA resources allowing more accelerators to be deployed. The fastest precise configuration includes one bilateral, one tracking and one integration kernel and runs at 2.91 fps, i.e. 9.4 times slower than the fastest approximate solution.

4.4 Power and Area exploration

The average power of the PL fabric for all our experiments is in the range (2.6W, 3.2W), whereas the average power of ARM with 4 threads in the range (1.27W, 1.65W). The average power dissipation in the fastest approximate FPGA configuration is 1.56W for ARM and 3W for the PL fabric. On the other side, the average power in the baseline implementation on x86 Xeon W-2123 is 51.4W. Table 4.8 shows the difference between the baseline x86 configuration and the fastest FPGA configurations in performance, error, power and energy consumption for the lr.kt2 trajectory.

Configuration	Mean Execution Time per Frame [ms] / FPS	RMSE [cm]	Mean Power [W]	Energy [J]
Baseline - x86	37.5 / 26.7	2	51.4	1722
Fastest Precise	343 / 2.9	2	1.4 / 2.7	424 / 817
Fastest Approximate	36.3 / 27.5	2.5	1.56 / 3	50 / 96

Table 4.8: System metrics for x86 and FPGA fastest configurations

Figure 4.9 shows the FPGA resource utilization for the fastest precise and the fastest approximate configurations. Approximation allows efficient use of resources by reducing the area of the bilateral filter and the integration kernels, thus, enabling multiple integration accelerators. Figure 4.10 shows that the increase in performance (frames per second) for the entire application requires more FPGA resources (BRAM, DSP, FF, LUT).

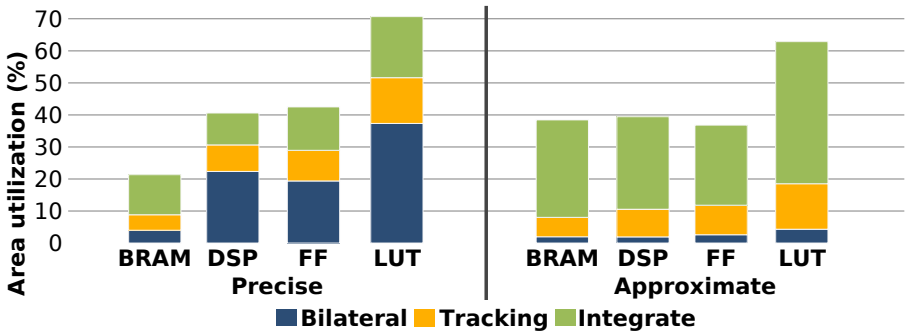

Figure 4.9: Area utilization for the fastest precise and approximate configurations.

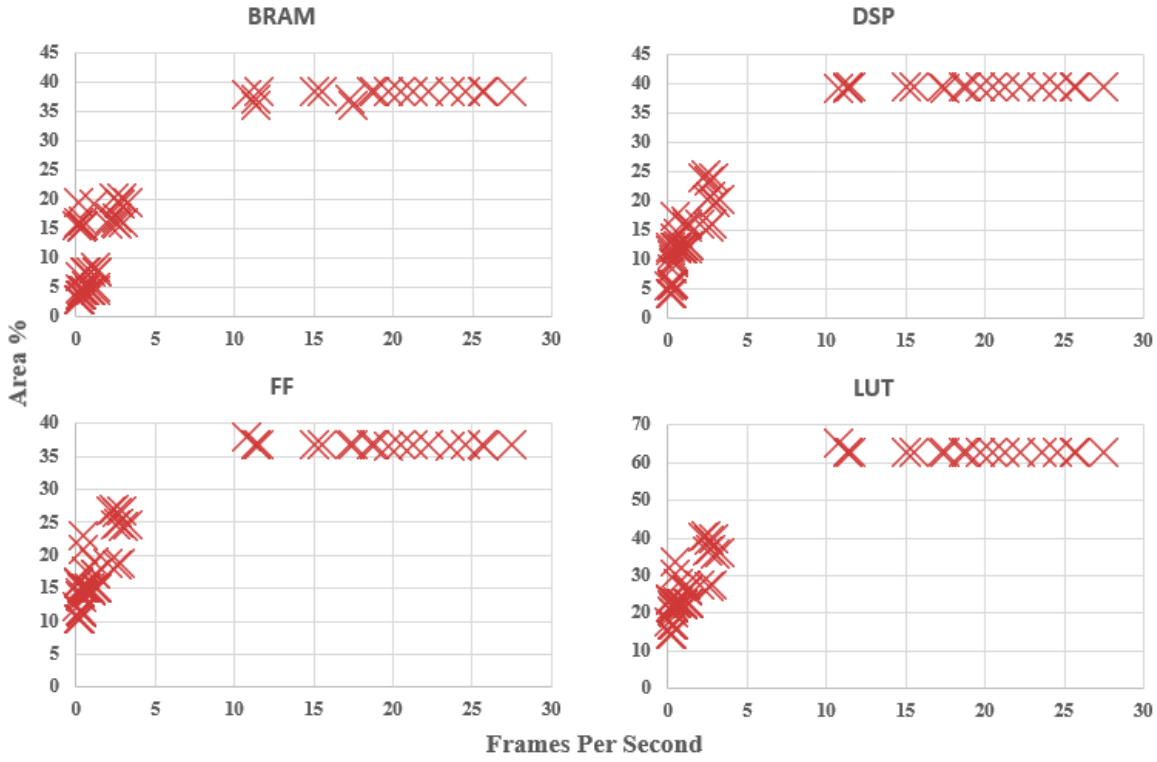

Figure 4.10: Area utilization vs. throughput for the entire application configurations.

4.5 Timeline analysis

The timelines of Figure 4.11 present the execution time and the RMSE per frame for the fastest precise and the fastest approximate FPGA configurations for two trajectories lr.kt1 and lr.kt2. The graphs reveal considerable execution time variation (especially in lr.kt2), in both the precise and approximate executions (the latter is not clearly visible in this graph due to lower values of execution time). The reduced execution time around frame 365 is due to the fast exits from the raycast and integration kernels since most objects in the scene are much closer to the moving agent than, for example, the objects in frame 460. Note also the ubiquitous ”high frequency” variations, which are almost entirely due to small intra-frame execution time variations of the tracking kernel.

Our fastest configuration achieves 29.5 fps (33.5ms per frame) in lr.kt1 and 27.5 fps (36.3ms per frame) in lr.kt2. The Fastest Approximate RMSE line in lr.kt1 shows higher error deviation from the baseline compared with the error deviation in lr.kt2 trajectory. However, since this value does not exceed 1.5cm and all frames are tracked, our configuration is acceptable.

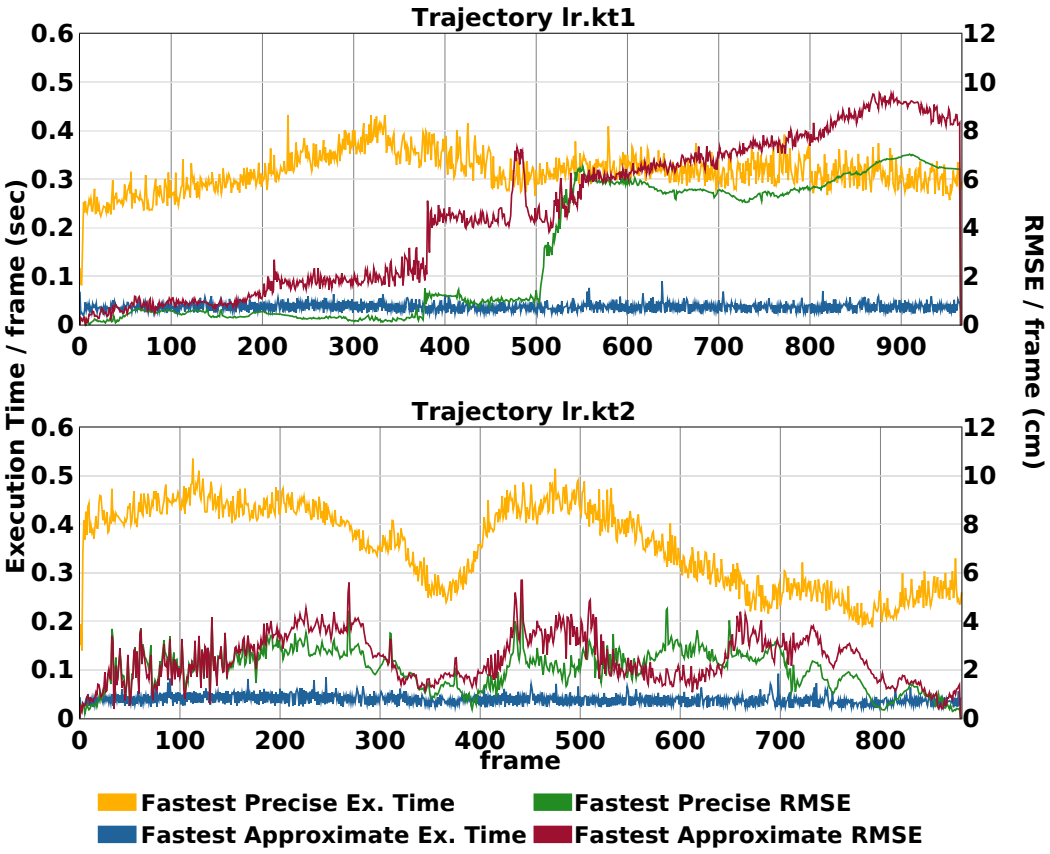

Figure 4.11: Timeline showing execution time and RMSE per frame for the fastest precise and approximate FPGA implementations.

Chapter 5

Conclusions

Dense visual SLAM algorithms have stringent throughput and accuracy requirements, a combination that makes high performance implementations particularly challenging, especially in power-constrained environments. In this thesis, we have described and evaluated a very large space of parameterizable MPSoC FPGA architectures for the KinectFusion algorithm by blending together precise and approximate optimizations. We have shown that even though approximations provide additional speedup on top of what is achieved by conventional hardware optimizations, they need to be judiciously applied to avoid large cumulative errors. We proposed a systematic methodology to rank the impact of each optimization on the performance and output error of KinectFusion, and used it as an optimization selection mechanism. Our best FPGA design achieve 27.5 fps (11.55x faster than the ARM OpenMP implementation and 1.03x faster than the Xeon W-2123 OpenMP implementation) at an 320x240 input depth frame resolution without exceeding the tight error bounds necessary for tracking convergence.

Bibliography

[1] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan, November 2007.

[2] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*, pages 2320–2327, 2011.

[3] Redhwan Jamiruddin, Ali Osman Sari, Jahanzaib Shabbir, and Tarique Anwer. Rgb-depth slam review. *arXiv preprint arXiv:1805.07696*, 2018.

[4] Carnegie Mellon University. Real-time dense 3d reconstruction from rgbd. `http://graphics.cs.cmu.edu/courses/15769/fall2016/lecture/realtime3d`.

[5] Bruno Bodin et al. SLAMBench2: Multi-Objective Head-to-Head Benchmarking for Visual SLAM. In *ICRA*, 2018.

[6] Xilinx zynq ultrascale+ mpsoc zcu102 hardware documentation. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#hardware`.

[7] Ethan Eade. Lie groups for 2d and 3d transformations, 2017. `http://www.ethaneade.org/lie.pdf`.

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

[9] 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. Introducing SLAMBench, a Performance and Accuracy Benchmarking Methodology for SLAM. In *International Conference on Robotics and Automation, (ICRA), Seattle, WA, USA, 26-30 May*, pages 5783–5790, 2015.

[10] Vitis Unified Software Platform. `https://www.xilinx.com/products/design-tools/vitis.html`.

[11] TJ Chong, XJ Tang, CH Leng, M Yogeswaran, OE Ng, and YZ Chong. Sensor technologies and simultaneous localization and mapping (slam). *Procedia Computer Science*, 76:174–179, 2015.

[12] Wikipedia. Monocular. `https://en.wikipedia.org/wiki/Monocular`.

[13] Á. P. Bustos, T. Chin, A. Eriksson, and I. Reid. Visual slam: Why bundle adjust? In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2385–2391, 2019.

[14] Christian Pirchheim, D. Schmalstieg, and Gerhard Reitmayr. Handling pure camera rotation in keyframe-based slam. *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 229–238, 2013.

[15] Jakob Engel et al. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *ECCV-Part II*, 2014.

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

[17] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[18] Ruben Gomez-Ojeda, David Zuñiga-Noël, Francisco-Angel Moreno, Davide Scaramuzza, and Javier Gonzalez-Jimenez. PL-SLAM: a Stereo SLAM System through the Combination of Points and Line Segments. *arXiv preprint arXiv:1705.09479*, 2017.

[19] Andreas Geiger, Julius Ziegler, and Christoph Stiller. Stereoscan: Dense 3d reconstruction in real-time. In *IEEE Intelligent Vehicles Symposium*, Baden-Baden, Germany, June 2011.

[20] Wikipedia. Kinect. `https://en.wikipedia.org/wiki/Kinect`.

[21] Richard A. Newcombe et al. KinectFusion: Real-time dense surface mapping and tracking. In *ISMAR*, 2011.

[22] Thomas Whelan, Stefan Leutenegger, Renato Moreno, Ben Glocker, and Andrew Davison. Elasticfusion: Dense slam without a pose graph. 07 2015.

[23] Angela Dai, Matthias Nießner, Michael Zollöfer, Shahram Izadi, and Christian Theobalt. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. *ACM Transactions on Graphics 2017 (TOG)*, 2017.

[24] C. Kerl, J. Sturm, and D. Cremers. Dense visual slam for rgb-d cameras. In *Proc. of the Int. Conf. on Intelligent Robot Systems (IROS)*, 2013.

[25] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John McDonald. Kintinuous: Spatially extended kinectfusion. 2012.

[26] V A Prisacariu, O Kähler, S Golodetz, M Sapienza, T Cavallari, P H S Torr, and D W Murray. InfiniTAM v3: A Framework for Large-Scale 3D Reconstruction with Loop Closure. *arXiv pre-print arXiv:1708.00783v1*, 2017.

[27] Luca Carlone, Roberto Tron, Kostas Daniilidis, and Frank Dellaert. Initialization techniques for 3d slam: A survey on rotation estimation and its use in pose graph optimization. *Proceedings - IEEE International Conference on Robotics and Automation*, 2015:4597–4604, 06 2015.

[28] Jose Luis Blanco. A tutorial on se(3) transformation parameterizations and on-manifold optimization. 09 2010.

[29] Carlo Tomasi and Roberto Manduchi. Bilateral Filtering for Gray and Color Images. In *ICCV, Bombay, India, Jan.*, 1998.

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

[31] Brian Curless and Marc Levoy. A Volumetric Method for Building Complex Models from Range Images. In *SIGGRAPH*, 1996.

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

[33] Berthold K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *J. Opt. Soc. Am. A*, 4(4):629–642, Apr 1987.

[34] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based heterogeneous FPGA architectures: application specific exploration and optimization, Chapter 2*. Springer Science & Business Media, 2012.

[35] Quentin Gautier et al. FPGA Architectures for Real-time Dense SLAM. In *ASAP*, Jul 2019.

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

[37] Quentin Gautier et al. Real-time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability trade-offs of the Altera OpenCL SDK. In *FPT*, 2014.

[38] Weikang Fang et al. FPGA-based ORB Feature Extraction for Real-Time Visual SLAM. *CoRR*, abs/1710.07312, 2017.

[39] Mohamed Abouzahir et al. Embedding SLAM Algorithms: Has it come of age? *Robotics and Autonomous Systems*, 100, 2018.

[40] Michael Montemerlo et al. FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges. In *IJCAI*, 2003.

[41] Konstantinos Boikos and Christos Savvas Bouganis. A Scalable FPGA-Based Architecture for Depth Estimation in SLAM. In *ARC*, 2019.

[42] Konstantinos Boikos and Christos Savvas Bouganis. A High-Performance System-on-Chip Architecture for Direct Tracking for SLAM. In *FPL*, 2017.

[43] Amr Suleiman et al. Navion: A 2-mW Fully Integrated Real-Time Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones. *IEEE Journal of Solid-State Circuits*, 54(4), 2019.

[44] Jinwook Oh, Jungwook Choi, Guilherme C. Januario, and Kailash Gopalakrishnan. 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.

[45] Yan Pei et al. SLAMBooster: An Application-Aware Online Controller for Approximation in Dense SLAM. In *PACT*, 2019.

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

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