FPGA Architectures for Real-time Dense SLAM

Quentin Gautier, Alric Althoff and Ryan Kastner
Department of Computer Science and Engineering
University of California, San Diego, USA
{qkgautier, aalthoff, kastner}@eng.ucsd.edu

***Abstract*— Simultaneous Localization And Mapping (SLAM) is an important technique used in robotics, computer vision, and virtual/augmented reality. SLAM algorithms have moved past creating sparse maps to making dense 3D reconstruction of the environment. Dense SLAM algorithms have high computational demands that require hardware acceleration to be done efficiently in real-time. FPGAs are an attractive compute platform for SLAM systems as they are low power and high performance. Unfortunately, dense SLAM algorithms are complex and FPGAs are notoriously difficult to program. In this work, we study the best techniques for accelerating 3D reconstruction on FPGA. We analyze a 3D reconstruction system, and implement modular FPGA designs for the main components of this application. We target both an FPGA SoC and a larger FPGA PCIe board, and perform a design space exploration (DSE) of our designs. We analyze the results of our DSE, characterize the design spaces to highlight important features, and we implement the best designs in an open-source and end-to-end dense SLAM system running on a FPGA SoC board. On the SoC board, using the FPGA increases the throughput of the whole application by a factor of two compared to the ARM processor, and individual algorithms are up to 38 times faster on the FPGA.**

I. Introduction

Simultaneous localization and mapping (SLAM) is a general technique that creates a spatial map of the environment while determining the position of an agent inside this map. SLAM has fundamental applications in robotics, computer vision, and virtual/augmented reality. Ideally, a SLAM system works in real-time, provides a detailed 3D map, uses minimal power, has a small physical footprint, and is low-cost. But these are often conflicting constraints that create a complex design space.

Early SLAM algorithms dialed back the algorithmic complexity to achieve suitable performance. For example, sparse SLAM algorithms consider a subset of sensor data and only model the environmental information needed for navigation [1]. This was largely in response to available resources, e.g., low-power compute platforms were not available, and sensors provided relatively low bandwidth information. As the efficiency of compute platforms increased, SLAM algorithms added the ability to model more complex environments. It is now possible to produce a real-time detailed 3D model of the world using dense SLAM aka 3D reconstruction algorithms. To work in real-time, these algorithms must process a high volume of information from large amount of sensor data (cameras, depth sensors, IMU, LIDAR, etc.). These systems often carefully leverage hardware acceleration techniques, e.g., by operating on GPUs and FPGAs [2], [3], [4].

Our ultimate goal is to determine the best way to implement dense SLAM using an FPGA-accelerated system. This requires us to perform architectural optimizations to carefully balance between resource usage, performance, and accuracy. However, the space of possible optimizations is vast, and very slow to explore; testing a single design can take multiple hours. As a result, we want to increase our understanding of the effects of each optimization in an effort to improve the design space exploration of similar applications.

We develop a set of highly parameterized architectures for each of the dense SLAM components (tracking, depth fusion, and ray casting). Each component is outfitted with multiple optimizations that can be tuned to offer tradeoffs between FPGA resource utilization, throughput, and quality of result. We compile thousands of unique designs based on these optimizations, and run each of them on two representative FPGA platforms. The first is the Terasic DE1 FPGA System-on-Chip (SoC), which is a low-cost, low-power integrated system with an ARM processor and a small FPGA. The second is the Terasic DE5 PCIe board, which has an FPGA that is approximately 7× larger than the FPGA on the DE1. We provide an analysis of the resulting spaces to better understand the complex relationship between the tuning parameters and the output architectures.

We develop the FPGA architectures using the Intel OpenCL SDK for FPGA, which provides the flexibility to perform high-level design tradeoffs and eases the integration into the existing dense SLAM frameworks. We develop a complete system capable of performing real-time dense SLAM (Fig. 1) which is guided by our design space analysis. Our designs, design space data, and analysis are made open-source [5] to facilitate follow-on work related to SLAM, FPGA design, and hardware design space exploration.

Our main contributions are to provide:

- Highly optimized OpenCL FPGA code for the major components of dense SLAM.
- A new algorithm combining two major components of dense SLAM.
- A comprehensive parameterization of these implementations to easily target different FPGA resources constraints and applications demands.
- A statistical analysis of the design space considering more than 2500 possible implementations with different resource utilization and performance.
- An open-source release of our algorithms and data.

In Section II we discuss related work on FPGA-accelerated SLAM. In Section III, we introduce the 3D Reconstruction

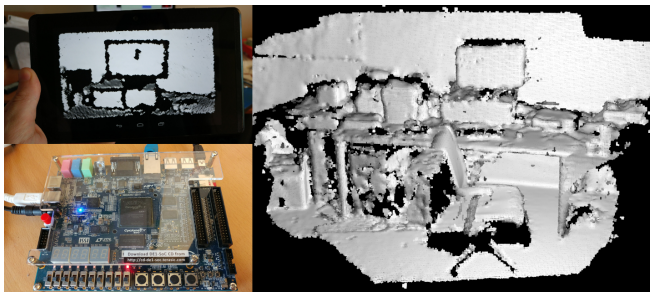

Fig. 1: 3D model reconstructed in real-time on a DE1 FPGA SoC with the data transmitted from a Google Tango tablet.

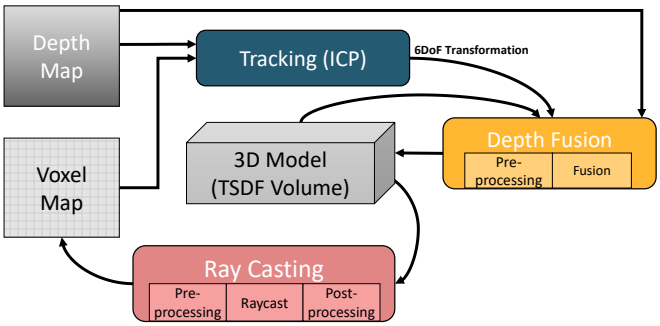

Fig. 2: InfiniTAM Reconstruction workflow. Each iteration of the algorithm processes an input depth map through three steps: ICP, Depth Fusion, and Ray Casting.

framework, and we detail its FPGA implementation in Section IV. Section V shows the results of our design space exploration, and we conclude in Section VI.

II. Related Work

SLAM is commonly deployed in applications that utilize high bandwidth sensors, require real-time results, and have a limited power budget [6]. This has naturally pushed SLAM designers towards hardware accelerated platforms.

FPGAs are a particularly attractive platform due to their power efficiency. For example, the bottleneck in visual SLAM algorithms (i.e., using cameras as sensors) is feature detection (detecting points of interest in an image) and feature extraction (encoding the visual features for distance calculation). Ulusel et al. [7] analyze one feature detection algorithm (FAST) and two feature extraction algorithms (BRIEF, BRISK) on embedded CPU, GPU, and FPGA. Feature detection and extraction are common tasks in SLAM. Their results show that the FPGA implementation outperforms the CPU and GPU in both power and performance.

The complexity of SLAM algorithms makes it difficult to implement an entire end-to-end system utilizing solely an FPGA. FPGA SoCs are an appealing option as the algorithm can be split across hardware and software. For example, Tertei and Devy [8] implement a version of SLAM based upon an Extended Kalman Filter. They perform matrix multiplication on the FPGA and the remainder of the algorithm in software. Nikolic et al. [9] build a visual-inertial motion estimation system. They offload the feature detection (Harris corners / FAST corners) onto the FPGA. Similarly, Aguilar-Gozalez et al. [10] describe an FPGA implementation of the detection/extraction process to increase the number of features detected by standard feature detection algorithms.

Other works utilize the FPGA for a larger portion of the application. The authors in [11] implement a particle filter SLAM on the FPGA. The input comes from a sparse laser scanner and the map is created as a simple occupancy grid. Another work implements a large portion of the Scan-Matching Genetic SLAM (SMG-SLAM) algorithm [12] on an FPGA. SMG-SLAM is similar to our algorithm (see Section III), but takes its input from a sparse laser range finder. The result is stored in an occupancy grid map with a resolution between 2 cm and 12 cm and a fairly low number of grid cells (up to 724). While occupancy grids are an efficient map representation for navigation purposes, other representations such as Signed Distance Function [13] are more adapted to dense 3D reconstruction. More recently, Boikos and Bougnais [14], [15] accelerate the *semi-dense* LSD-SLAM algorithm on an FPGA SoC achieving 22 frames per second on a 320x240 input visual frame. We are able to handle denser environmental maps than these projects.

Dense SLAM algorithms focused on creating detailed 3D models are often classified as *real-time 3D reconstruction* algorithms. There are only a few recent attempts to port 3D reconstruction algorithms to an FPGA. Gautier et al. [2] implement the KinectFusion algorithm [16] on an FPGA. KinectFusion performs dense 3D reconstruction based on an input from a RGB-D sensor. They were only able to successfully implement one portion of the 3D reconstruction on FPGA; the other algorithms could not be integrated due to limited memory bandwidth. We are able to handle all the algorithms. Additionally, their work also targets a single large FPGA. It does not consider the appropriate trade-offs that could be made to use smaller FPGA SoC systems.

III. Dense SLAM Overview

We create FPGA architectures for the SLAM framework called *InfiniTAM* [4] which is itself derived from KinectFusion. InfiniTAM runs on a multicore CPU or on a GPU. Fig. 2 illustrates the InfiniTAM 3D Reconstruction framework. It takes as input depth maps from an RGB-D camera (e.g., the Microsoft Kinect) and iteratively processes these depth maps through three main modules: **Tracking**, **Depth Fusion**, and **Ray Casting**, and updates a global **Map** that represents the 3D model. Below we give a brief description of these modules; exact details can be found in the original paper [4].

1) Map Representation: The 3D model is stored in a grid of voxels using the Truncated Signed Distance Function (TSDF) [13]. Each voxel records the distance to the nearest surface, along with a confidence value. The distance is normalized to a maximum value called *truncation distance*. An efficient way of storing a TSDF volume is to use a hash table that only keeps non-empty voxels. Hash tables are usually very efficient on FPGAs [17], [18]. InfiniTAM leverages a voxel hashing representation [19] with a hash

function based on the coordinates. The hash table actually references *blocks* of $8 \times 8 \times 8$ voxels, while the actual voxels are stored in a flat memory buffer.

2) Depth Fusion: Depth Fusion starts with a pre-processing step to calculate the visibility of each block from the camera position. The main algorithm iterates over the voxels in the precomputed visible blocks. Each voxel is projected into the depth map, and its distance to the camera center is compared to the depth map value. If the two distances are within the truncation threshold, they are combined using a weighted average.

3) Ray Casting: Ray Casting also starts with a pre-processing stage to create a rough bounding box around visible blocks. Then, the main computation consists of sending a ray from each pixel of a 2D view into the 3D TSDF model within the bounding box, to find a point where the sign of the distance changes. The result is saved as a 3D point in a Voxel Map. A post-processing step estimates a surface normal for each resulting 3D point.

4) Tracking: Tracking is based on the Iterative Closest Point (ICP) algorithm [20]. ICP finds the optimal alignment between the input depth map and the 2D projection of the current model from the last camera position (the output of Ray Casting). The transformation between these two inputs represents the camera motion between frames. The details of this particular implementation of ICP are presented in the original Kinect Fusion paper [16], with an additional option in InfiniTAM to solve for rotation only or translation only.

IV. FPGA Architecture

In this section, we describe our proposed FPGA architectures for the main components of InfiniTAM, along with the optimization options that we enable. We focus on the core algorithms, without pre- or post-processing steps.

For each algorithm, we develop parameterized OpenCL code that is synthesized to an FPGA using the Intel FPGA OpenCL SDK. Different optimizations are enabled, disabled, and tuned using parameters, also called *knobs* for design space exploration purposes [21], [22]. We define many knobs (e.g., loop unroll factor, memory layout, etc.) that generate designs with different performance, resource usage, and accuracy. In the following, we describe the OpenCL implementations for each algorithm, with a focus on the key optimization knobs that we analyze in Section V.

We also propose a new algorithm that combines Depth Fusion and Ray Casting to improve throughput. We refer to this algorithm as the *Combined Kernel*.

A. Depth Fusion

The Depth Fusion algorithm takes as input the depth map, a list of hash table entries indexes, the hash table, and the buffer containing actual voxel data. The algorithm contains an outer loop to process each hash table entry (block), and an inner loop to process the 512 voxels inside each block. In the following, we describe how we modified this implementation with FPGA-specific optimizations.

Loop Optimizations: The OpenCL compiler offers multiple options to implement loops. This includes using OpenCL *work-items* (WI) (on FPGA, a WI is generally interpreted as a single – data-dependency free – stage of a pipelined loop), a simple *for* loop that automatically gets pipelined, or *work-groups* (groups of WI) that enable coarse-grained parallelism. We provide knobs to switch between these representations. The coarse-grain parallelism enabled with work-groups is an OpenCL feature called *compute units*. Each compute unit is a duplication of the kernel to increase the parallelism and thus easily scale the task on larger devices. We provide another knob to control the number of compute units.

Memory Optimizations: Input buffers can be cached using *local memory*, which has better latency for non-predictable accesses, such as depth map indexing. We implement a knob to optionally pre-load the depth map into local memory in a predictable way. In this case, the depth size must be fixed (another knob) and the design uses more BRAMs. Lastly, to prevent a potential memory bottleneck when accessing voxels, we implement a knob to cache one block of voxels in local memory for reading and optionally for writing.

Additional Optimizations: We also provide knobs to control the unrolling factors on various loops. Another knob controls the placement of a branch condition, which either groups voxel accesses together or keeps an early branch to potentially terminate the loop earlier.

B. Ray Casting

The Ray Casting algorithm iterates over the pixels of a 2D view to project 3D information into that view. For each pixel, it steps along a 3D ray and reads the distance value from the hash table until that distance becomes negative. Then the algorithm refines the location of the surface by stepping backwards along the ray using interpolated distance values. Estimations of the starting and ending points of the ray are pre-calculated in a downsampled *min/max* input map. The output is a *Voxel Map*, an organized point cloud containing one 3D point per pixel.

Memory Optimizations: The data access pattern is complex and non-contiguous particularly when fetching voxel data. Consecutive accesses to the same data can be manually cached, and the compiler also provides an automatic cache mechanism. We provide knobs to enable/disable these caches, which provides tradeoffs between BRAM usage and cache speedup. The interpolation fetches 8 different values for averaging. We provide the option to disable this interpolation, which may decrease the overall quality of the SLAM system. Finally, the algorithm ends with a *Refine* step that creates another voxel data access. We can disable it with a knob. These last two knobs are often necessary to save logic elements and fit the kernel on smaller devices.

Other Optimizations: Just like Depth Fusion, we can also switch between a *for* loop and work-items for the outer loop, and we have the option to fix the 2D image size to simplify some calculations. We can also remove some coordinate system transformations and memory accesses by simplifying the start and end points of the ray. The ray can start at a

Algorithm 1: Combined Algorithm

Input : Depth Map; Visible blocks IDs; Hash table; Voxel Buffer; Truncation threshold μ

1 **for** *each visible block ID* **do**
2 Fetch block B from hash table
3 From B, fetch pointer to voxel block in voxel buffer
4 **for** *each voxel* $V = (x, y, z)$ *in the voxel block* **do**
5 $(g_x, g_y, g_z) \leftarrow$ Calculate global coordinates of V
6 $(c_x, c_y, c_z) \leftarrow (g_x, g_y, g_z)$ to camera view
7 $(i, j) \leftarrow$ Project (c_x, c_y, c_z) into 2D image
8 $D \leftarrow$ Get depth at (i, j)
9 $(w, d) \leftarrow$ Fetch weight and distance from V
10 **if** $w \leqslant w_{max}$ *or* $|c_z - D| \leqslant \mu$ **then**
11 $(w', d') \leftarrow$ New weight and distance
12 Store new voxel $V \leftarrow (w', d')$
13 **end**
14 $d \leftarrow$ Fetch distance from V
15 **if** $|d| <$ *ProjectionMap[i,j]* **then**
16 $c_z \leftarrow c_z + d$
17 $(x, y, z) \leftarrow (c_x, c_y, c_z)$ to global coord.
18 Save (x, y, x) position into Voxel Map
19 ProjectionMap[i,j] $\leftarrow |d|$
20 **end**
21 **end**
22 **end**

depth of 0, and end at the maximum sensor range, which makes an assumption about the input data. We keep both options as knobs.

C. Combining Depth Fusion and Ray Casting

Ray Casting provides a view of the fused 3D model at the current position, and as such, accesses most of the voxels that have just been updated by Depth Fusion. Thus, it is beneficial to combine both steps into one coordinated process. However these two steps process data in opposite directions. Depth Fusion projects 3D voxels into a 2D view, while Ray Casting sends rays from the 2D view into the 3D volume. Depth Fusion is more efficient as it involves less memory accesses due to the pre-processing step. We create a *Combined* kernel by integrating Ray Casting into Depth Fusion. We implement the uninterpolated and unrefined version of Ray Casting to avoid significant random accesses to the hash table and the voxel data. The main issue arises when Depth Fusion projects multiple points on the same pixel. Because we lose the sequential aspect of the Ray Casting algorithm, we need a way to select which distance (d) to keep on each pixel.

The Combined algorithm is presented in Algorithm 1. There are two computation blocks: the Depth Fusion block at line 10, and the Ray Casting block at line 15. The Depth Fusion is similar to the orginal implementation. The Ray Casting block writes the distance d of the current voxel to a *projection map* which is the same size as the 2D image view. The goal of the projection map is to save only the smallest d in the projected pixel. When a new distance d_{new} is projected into the map, it is saved only if $|d_{new}| < |d|$, at which point the Voxel Map is updated with the new point. This creates a dependency between iterations that is difficult to avoid without duplicating the entire projection map.

Optimizations: This implementation mostly uses the same knobs as Depth Fusion with some restrictions. We restrict the outer and inner loops to be implemented as *for* loops only. We add the possibility of implementing the inner loop as work-items, which in the OpenCL model creates a race condition on the projection map. On FPGA, this race condition occurs depending on the depth of the pipeline, which can create a small loss of precision in the result. We also add a knob to use either local or external memory for the projection map.

D. Iterative Closest Point (ICP)

ICP aligns the current depth map with the Voxel Map from Ray Casting. It finds pairs of corresponding points by projecting them into a common 2D frame and rejecting candidate pairs based on threshold values. The distance and angle between pairs of points are turned into a system of equations represented by two triangular matrices H and ∇. These matrices are summed together over all points and solved on the CPU to update a global transformation matrix. ICP has the option to solve for rotation or translation only, which uses smaller H and ∇ matrices. We propose a number of optimizations for the FPGA implementation.

Interpolation: The 3D points and 3D normals from the depth map and Voxel Map can be read with bilinear interpolation. These interpolations can be disabled with knobs.

Branching: This kernel has a lot of branches and therefore a lot of control logic. It contain branches to differentiate between *long* and *short* iterations, i.e., whether we solve for rotation and translation, or only one component. We implement a knob to disable these branches, and fully calculate the matrices even for short iterations. This helps reduce the resource utilization.

Accumulation: This kernel has a lot of data dependency between each iteration due to the matrices needing to be summed together. We implement multiple shift registers to decrease this dependency. To balance between speed and logic utilization, we provide knobs to enable shift registers and control their size. In the case where we do not use shift registers, we implement knobs which unroll the computation and accumulation of the matrices to decrease the latency.

V. Experimental Results And Analysis

In this section, we analyze the effects of the various hardware optimizations to provide insight on how to design architectures for real-time dense SLAM systems. To ground our design process in reality, we focus on two different FPGA platforms – an embedded FPGA SoC and a higher performance PCIe FPGA system.

A. Experimental Setup

We implement the full end-to-end dense SLAM system on: 1) a Terasic DE1-SoC board with a Cyclone V FPGA

TABLE I: Summary of the knobs for all four algorithms, along with their names used in Section V

Depth Fusion		Ray Casting		ICP		Combined	
Name	Description	Name	Description	Name	Description	Name	Description
ComputeUnits	Num. compute units	**Interpolate**	Enable interpolation	**PointInterpolation**	Point interpolation	**-**	Same knobs as
XyzLoop	Inner loop type	**Refine**	Enable refine step	**NormalInterpolation**	Normal interpolation		Depth Fusion
EntryIdLoop	Outer loop type	**IndexCache**	Voxel access cache	**NablaSumtype**	How ∇ is summed		
EntryIdNumWI	Outer loop num. WI	**HashCLcache**	Hash access cache	**HessianSumtype**	How H is summed	**SdfLocal**	Projection map
HardcodeSize	Hardcode depth size		(auto)	**Branch**	Enable branching		cache
CacheVoxels	Voxels block cache	**VoxelCLcache**	Voxel access cache	**NablaUnroll**	∇ sum - unroll factor		
XyzLoopFlat	Flatten inner loop		(auto)	**HessianUnroll**	H sum - unroll factor		
XyzUnroll	Unroll inner loop	**HardcodeSize**	Hardcode image size	**ShiftRegister**	Size of shift registers		
EntryIdUnroll	Unroll outer loop	**UseWi**	Outer loop type				
DepthLocal	Depth map cache	**Minmax**	Simplify start/end				
BranchPos	Condition placement		of ray				

and a dual-core ARM Cortex A9 processor, and 2) a Terasic DE5 PCIe board with a Stratix V FPGA, connected to a workstation with an x64 quad-core i7-4790K CPU.

We write all of our kernels using OpenCL, compile them with the Intel FPGA OpenCL SDK v16.1, and integrate them into the *InfiniTAMv2* code base. Each combination of knobs in our kernels produces a unique design. We choose reasonable values for knobs by selecting mostly powers of two and values likely to generate a small design, considering the smallest FPGA. In total, we have compiled more than 480 unique designs across all algorithms for the DE1 FPGA, and more than 2600 for the DE5 FPGA.

For most of our experiments, we fix the input depth map size to *QVGA* (320x240), with a few designs compiled for an input size of 320x180. To test and measure the performance of the kernels, we use the following benchmarks from the standard *TUM* RGB-D SLAM dataset [23]: **fr1/desk**, **fr1/360** and **fr1/room**. We also use a custom dataset (**Cave**) made from data collected with a Google Tango tablet in an underground cave environment. We set the TSDF voxel size to 1 cm, which is comparable to the depth resolution of the sensor used in the benchmarks.

We record the running time of different modules of InfiniTAM. Due to the limited amount of shared memory that can be allocated on the SoC board, our kernels are configured to handle a limited amount of data in the 3D model. We run each benchmark for a specific number of frames (fr1/desk: 595; fr1/room: 700; fr1/360: 440; cave: 400). Below we present the results of this design space exploration, combining FPGA logic utilization and running time.

B. FPGA SoC Design

Here we describe a fully functional dense SLAM system running on a small Cyclone V FPGA SoC. Consequently, we want to focus on designs that have low resource usage while still maintaining real-time performance. To do this, we perform comprehensive analysis of the design spaces for the different dense SLAM components.

Fig. 3 illustrates the design spaces on one of our benchmarks (room). Fig. 3(a) shows the total frame rate of the application when running different designs of different modules on FPGA, and Fig. 3(b) shows the throughput measured individually on each design.

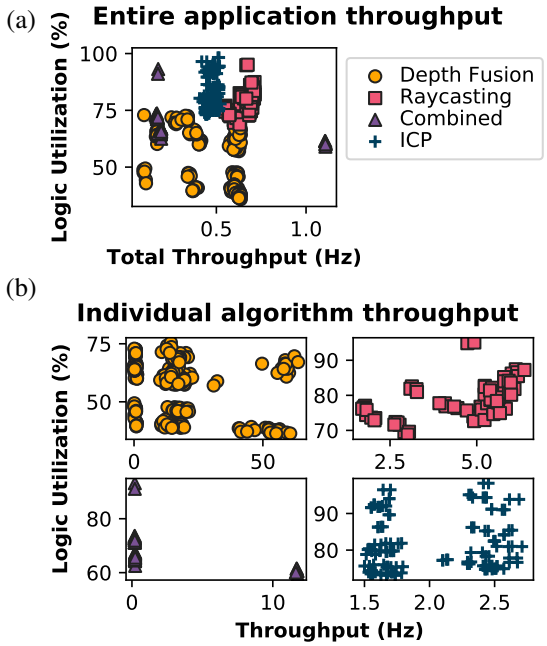

Fig. 3: Design spaces of InfiniTAM running on the DE1 board with the *Room* benchmark. (a) plots the throughput of the entire application when running the selected algorithm on FPGA; (b) plots the throughput of individual algorithms. The y axis shows the logic utilization of individual algorithms.

ICP has the worse individual throughput overall and does not contribute to decrease the total running time of the application. The best versions of Depth Fusion and Raycasting both increase the total throughput and the Combined algorithm provides the highest total throughput. This highlights the benefit of combining Depth Fusion and Ray Casting into a single kernel as the individual kernels cannot both fit on the device.

Throughput: Table II presents the lowest running times for each benchmark, on the dual-core ARM processor with OpenMP (*Baseline*) and with one module accelerated on FPGA. We use *RaycastO3*, which is the Ray Casting algorithm without the interpolation and refine steps. As we implement these optimizations on FPGA, we also transpose them to CPU for a fair comparison. The Combined algo-

TABLE II: Comparison of the lowest running times of different kernels running on the DE1 SoC board. Each row compares the algorithm running on the ARM processor of the DE1 (top, in ms) and on the FPGA (bottom, in ms).

	Cave	Desk	360	Room
Depth Fusion	702.25 18.09 (**38.8**×)	425.75 13.47 (**31.6**×)	573.90 16.80 (**34.2**×)	508.89 15.71 (**32.4**×)
RaycastO3	522.66 35.21 (**14.8**×)	538.92 30.49 (**17.7**×)	455.15 26.53 (**17.2**×)	448.64 28.34 (**15.8**×)
Combined	1224.91 78.29 (**15.6**×)	964.67 89.41 (**10.8**×)	1029.05 85.08 (**12.1**×)	957.53 84.92 (**11.3**×)
ICP	348.35 265.84 (**1.3**×)	537.88 357.57 (**1.5**×)	508.22 330.29 (**1.5**×)	557.56 368.87 (**1.5**×)

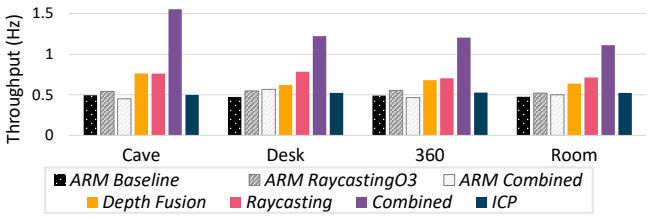

Fig. 4: Comparison of the average total frame rate between different versions of the application running on the DE1: running on ARM only, or the best versions accelerated on the FPGA. We present the results for different benchmarks.

rithm is compared to the added time of Depth Fusion and RaycastO3 on ARM.

The Depth Fusion algorithm presents the highest acceleration (up to 38×, or 34× for the TUM benchmarks). ICP is not very well accelerated because the designs implementing shift registers could not fit on this device, but the Combined kernel, replacing two of the three main algorithms, is accelerated more than 10×.

Fig. 4 shows the average throughput in frames per second (FPS) for each benchmark. We compare the different baseline versions (without modifications, with RaycastingO3, with the Combined algorithm) and the fastest FPGA versions. We achieve the best throughput with the Combined kernel, but Depth Fusion and Ray Casting are also faster than the baseline. We can achieve up to 1.55 FPS at 320x180, and up to 1.22 FPS at 320x240, which is faster than the CPU version (up to 0.49 and 0.65 FPS respectively).

LASSO Analysis: We perform a statistical analysis to better understand the impact of knobs on each design space. We use the LASSO operator [24] which creates a least square regression model over the knob values that best fits the objective curve (throughput, logic). LASSO can create sparse models by forcing knobs with a small contribution to the model to zero with an alpha parameter (a larger alpha will force more coefficients to zero). We perform a cross-validated search for the alpha that yields a model with the smallest Mean Squared Error (MSE). The result of this analysis is a coefficient for each knob, which represents a relative contribution for this knob to the linear model. A

TABLE III: LASSO analysis of throughput on the DE1 SoC board for the *Room* benchmark. We take the models with the minimum mean squared error (MSE), and show the 5 knob features with the largest contribution to that model. The knob names are summarized in Table I.

Depth Fusion (Min MSE = 0.0075)		**Raycasting** (Min MSE = 0.0010)	
Knob	Coef	Knob	Coef
$XyzLoop^2$	0.130	UseWI	0.261
$CacheVoxels^2$	0.067	$VoxelClcache^2$	0.152
CacheVoxels	0.031	Minmax	0.075
$XyzUnroll^2$	0.031	IndexCache,UseWi	0.071
CacheVoxels,XyzLoopFlat	-0.023	$HashClcache^2$	0.055
ICP (Min MSE = 0.0054)		**Combined** (Min MSE = 10^{-6})	
HessianUnroll	0.336	$XyzLoop^2$	0.155
NablaUnroll	0.040	XyzLoop	0.051
Branch NablaSumtype	0.023	XyzLoop,VoxelCache	-0.001
NablaSumtype	0.022	XyzFlat	0.001
HessianUnroll,NablaSumtype	0.016	$VoxelOutUnroll^2$	-0.001

larger coefficient indicates a higher correlation between the knob value and the constructed model. We focus our analysis on the throughput factor and we only present a summary of the results. Our complete data and scripts for a deeper analysis is available in our repository.

For each algorithm, we choose as inputs all the knobs and their 2^{nd} degree polynomial interactions to account for the non-linearity of certain knobs. We compute the model for multiple alpha, and find the model with the best MSE. Using the knobs without polynomial interactions leads to a higher MSE for most design spaces, which indicates a certain degree of non-linearity in the influence of knobs over the hardware architecture. We present the knobs with the highest coefficients for the models with the lowest MSE in Table III.

The dominant knobs are directly related to the implementation of a loop (*XyzLoop* in Depth Fusion / Combined; *UseWI* in Ray Casting). Using work-items has a great influence on the throughput when compared to using a for loop. The compiler is able to infer a better pipeline with work-items without dependency between iterations and an minimal initiation interval. Other important knobs are generally related to memory caching. Depth Fusion, Raycasting, and Combined all benefit from caching depth or voxel data in the local memory and these mechanisms have a large impact on the running time. The ICP and Combined algorithms are notably different. In the case of ICP, using shift registers to sum large vectors together requires a large amount of resources, and the designs using this optimization cannot fit on the small FPGA. As a result, the remaining designs are very sensitive to the amount of unrolling that is used to paralellize the vector accumulation. This situation leads to a much slower running time (Table II). The Combined kernel is also quite complex and certain memory optimizations do not fit on the Cyclone V FPGA. Consequently, the run time is largely dominated by the loop implementation knob: using work-items speeds up the algorithm, while other knobs impact the running time very little.

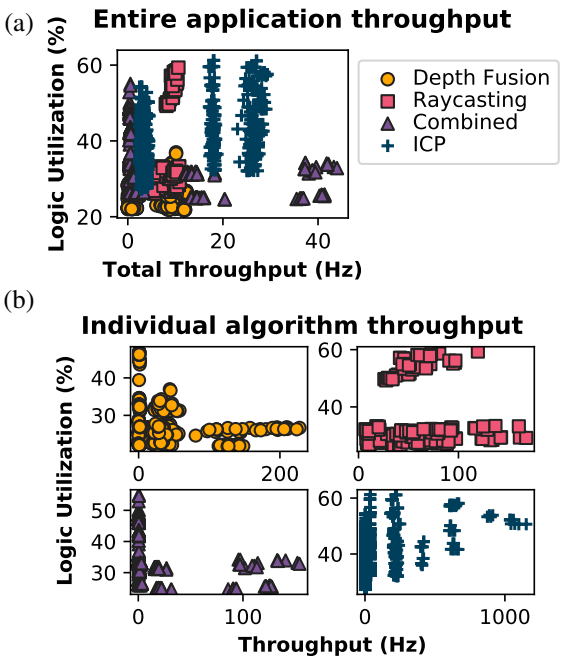

Fig. 5: Design spaces of InfiniTAM running on the DE5 board with the *Room* benchmark.

C. PCIe FPGA Design

We compile the algorithms on a DE5 board with a Stratix V FPGA, which is about 7 times larger than the DE1 FPGA (comparing *Logic Elements*, as defined by Intel). We perform a design space exploration to understand how the algorithms scale to a larger device, especially with respect to runtime.

Fig. 5 shows the design spaces of individual algorithms and of the entire application running on the *Room* benchmark. The differences with the DE1 design spaces are mostly due to the designs that do not fit on the smaller device. The logic utilization is generally low because most of the optimizations were focused on reducing this value to fit on the SoC chip. ICP produces a particularly different design space. There is a clear tradeoff between running time and logic utilization, which means that the most efficient designs have a high utilization and cannot be implemented on a smaller FPGA. The ICP algorithm clearly takes advantage of the increased number of registers on the Stratix V FPGA. Because it is computation-bound, it can be drastically sped up by using a large number of shift registers. Fig. 5(a) highlights the most efficient design tradeoffs for the entire application. While the application is not optimized for the DE5 board, we can see that using the Combined designs is generally faster.

We also compare the throughput improvement from the best designs on the DE1 to the designs on the DE5. We extend the design space exploration of individual kernels and compile two other types of bitstreams: one containing Depth Fusion, Ray Casting and ICP (DF+RC+ICP), and one with the Combined kernel and ICP (Combined+ICP). A full design space exploration of multiple kernels would lead to millions of unique combinations, so we only compile a few combinations of Pareto-optimal designs. We report the best performance of each algorithm for the different types of bitstreams (DF+RC+ICP, Combined+ICP), and individual algorithms in Fig. 6. In this figure, we compare the best throughput results between the DE1 FPGA, DE1 processor, DE5 FPGA, and DE5 processor, for individual algorithms and the entire application on the *Room* benchmark. Depth Fusion is improved by 3.6×, Ray Casting 4.8×, Combined 13.1×, and ICP is accelerated by 425.2×. The entire application runs at up to 44 FPS.

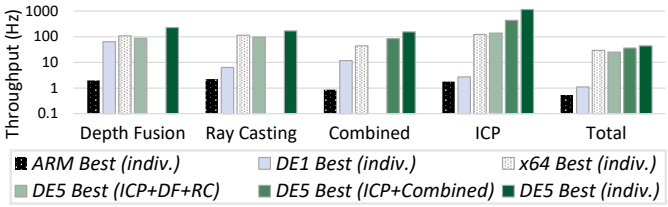

Fig. 6: Comparison of the best frame rate for individual algorithms and the entire application. We compare the best results on both DE1 and DE5 hardware setups, including the processor and FPGA results. On the DE5, we use bitstreams implementing either multiple algorithms (ICP+DF+RC / ICP+Combined) or only one (indiv.).

TABLE IV: LASSO analysis of throughput on the DE5.

Depth Fusion (Min MSE = 0.010)		**Raycast** (Min MSE = 0.003)	
Knob	Coef	Knob	Coef
$XyzLoop^2$	0.047	$HashCLCache^2$	0.152
XyzLoop,DepthLocal	-0.039	UseWI	0.116
ComputeUnits,DepthLocal	-0.039	Minmax	0.088
EntryidLoop,DepthLocal	-0.034	$IndexCache^2$	0.085
CacheVoxels	0.029	$Minmax^2$	0.055
ICP (Min MSE = 0.007)		**Combined** (Min MSE = 0.0016)	
HessianSumtype	0.056	$XyzLoop^2$	0.061
NablaSumtype	0.054	SdfLocal,XyzLoop	-0.024
NablaSumtype,ShiftRegister	0.047	SdfLocal,XyzUnroll	-0.012
HessianUnroll	0.038	SdfLocal,XyzFlat	0.012
HessianUnroll,HessianSumtype	0.033	XyzUnroll	-0.010

LASSO Analysis: We perform a LASSO analysis on these design spaces to better understand the importance of knobs on larger and more complex architectures. Table IV presents the summary of our LASSO analysis on the *Room* benchmark. In general, the throughput is harder to model on these larger spaces. There are more knob values and interactions, and as a result, the MSE is slightly higher for all algorithms. The general principles defined from the DE1 LASSO analysis still apply, as the type of loop implementation still has a great influence on the throughput, and memory caching is another useful optimization. The Combined algorithm has a large data dependency which is removed when using work-items, which tends to change the accuracy result. ICP is much less memory bound, and this fact is reflected in the knob coefficients, which tend to give weights to the knobs affecting the summation of data.

D. Real-Time Experiments

To test our implementation in a real environment using a complete system, we use a Google Tango tablet as both a depth camera and a screen to display real-time feedback. The Google Tango implements a light network client that only sends depth data and receives an image for display. The DE1 board receives, processes, and sends the data back through the wired/wireless network. We run the fastest Combined kernel from our design space exploration on the FPGA. We scan an office desk for about 1 minute (100 frames) and obtain the result shown in Fig. 1. In average, each frame took 505 ms to process (almost 2 FPS).

VI. Conclusion

We have described implementations of dense SLAM on an FPGA, outlining the potential hardware optimizations for the sub-algorithms (Tracking, Depth Fusion, and Ray Casting). We parameterized them to create a vast set of FPGA architectures, and analyzed the resulting design spaces to adapt them to two different FPGA platforms. We have built functional dense SLAM systems on two FPGA platforms. Our complete end-to-end system on the FPGA SoC achieves up to 2 FPS, and 44 FPS on a higher performance PCI-e FPGA. We expect that these numbers would be largely increased by using newer SoC boards currently on the market.

Design space exploration has provided insightful information about hardware tradeoffs for dense SLAM. Our analysis showed that the representation of loops in OpenCL for FPGA has a profound impact on the run time. Additionally, it benefits greatly from memory cache optimizations.

We have made our entire project open-source, which includes our full system implementations, OpenCL code, and design space exploration results. We hope that it provides the community a way to expand these results to more types of FPGA hardware, gain useful knowledge in creating hardware accelerated dense SLAM applications, and facilitates research on FPGA design space exploration tools.

References

[1] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "Monoslam: Real-time single camera slam," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1052–1067, June 2007.

[2] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "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)*, Dec 2014, pp. 326–329.

[3] T. Whelan, S. Leutenegger, R. Salas-Moreno, B. Glocker, and A. Davison, "Elasticfusion: Dense slam without a pose graph." Robotics: Science and Systems, 2015.

[4] O. Kahler, V. Adrian Prisacariu, C. Yuheng Ren, X. Sun, P. Torr, and D. Murray, "Very high frame rate volumetric integration of depth images on mobile devices," *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, no. 11, pp. 1241–1250, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2015.2459891

[5] (2019) Kernel source and results. [Online]. Available: https://github.com/KastnerRG/infinitam_fpga

[6] M. Johnson-Roberson, O. Pizarro, S. B. Williams, and I. Mahon, "Generation and visualization of large-scale three-dimensional reconstructions from underwater robotic surveys," *Journal of Field Robotics*, vol. 27, no. 1, pp. 21–51, 2010. [Online]. Available: http://dx.doi.org/10.1002/rob.20324

[7] O. Ulusel, C. Picardo, C. B. Harris, S. Reda, and R. I. Bahar, "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[8] D. T. Tertei, J. Piat, and M. Devy, "Fpga design and implementation of a matrix multiplier based accelerator for 3d ekf slam," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6.

[9] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P. T. Furgale, and R. Siegwart, "A synchronized visual-inertial sensor system with fpga pre-processing for accurate real-time slam," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 431–437.

[10] A. Aguilar-González, M. Arias-Estrada, and F. Berry, "Robust feature extraction algorithm suitable for real-time embedded applications," *Journal of Real-Time Image Processing*, Jun 2017. [Online]. Available: https://doi.org/10.1007/s11554-017-0701-8

[11] B. G. Sileshi, J. Oliver, R. Toledo, J. Gonçalves, and P. Costa, "Particle filter slam on fpga: A case study on robot@factory competition," in *Robot 2015: Second Iberian Robotics Conference*, L. P. Reis, A. P. Moreira, P. U. Lima, L. Montano, and V. Muñoz-Martinez, Eds. Cham: Springer International Publishing, 2016, pp. 411–423.

[12] G. Mingas, E. Tsardoulias, and L. Petrou, "An fpga implementation of the smg-slam algorithm," *Microprocess. Microsyst.*, vol. 36, no. 3, pp. 190–204, May 2012. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2011.12.002

[13] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/237170.237269

[14] K. Boikos and C. S. Bouganis, "Semi-dense slam on an fpga soc," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

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

[16] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, Oct 2011, pp. 127–136.

[17] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[18] D. Tong, S. Zhou, and V. K. Prasanna, "High-throughput online hash table on fpga," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 105–112.

[19] M. Niessner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Trans. Graph.*, vol. 32, no. 6, pp. 169:1–169:11, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508363.2508374

[20] P. J. Besl and N. D. McKay, "A method for registration of 3-d shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 239–256, Feb 1992.

[21] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–7.

[22] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, "Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on fpgas," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 918–923.

[23] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012, pp. 573–580.

[24] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996. [Online]. Available: http://www.jstor.org/stable/2346178