FPGA Architectures for Approximate Dense SLAM Computing

Maria Rafaela Gkeka, Alexandros Patras, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas
Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
{margkeka,patras,cda,lalis,nbellas}@uth.gr

***Abstract*—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. In particular, *dense* SLAM algorithms construct and update the map at pixel granularity at a high computational and energy cost especially when operating under real-time constraints.**

Dense SLAM algorithms can be approximated, however care must be taken to ensure that these approximations do not prevent the agent from navigating correctly in the environment. Our work introduces and evaluates a plethora of embedded MPSoC FPGA designs for KinectFusion (a well-known dense SLAM algorithm), featuring a variety of optimizations and approximations, to highlight the interplay between SLAM performance and accuracy. Based on an extensive exploration of the design space, we show that properly designed approximations, which exploit SLAM domain knowledge and efficient management of FPGA resources, enable high-performance dense SLAM in embedded systems, at almost 28 fps, with high energy efficiency and without compromising agent tracking and map construction. An open source release of our implementations and data can be found in [1].

***Index Terms*—Simultaneous Localization and Mapping, FPGA, approximate computing, energy efficiency**

I. 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. For this reason, 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 implementation very challenging, especially for an embedded system.

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 [2].

For our study, we use the KinectFusion algorithm of the SLAMBench suite [3]. Starting from the baseline C++/OpenMP implementation, we develop precise as well as approximate hardware accelerators for the most important components of the algorithm, by re-writing the code and using HLS directives.

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. We show that 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.

Section II introduces KinectFusion and section III describes the parameterized architectures and the approximations for each kernel. Section IV presents and discusses the results of our experimental evaluation. Section V discusses related work. Finally, section VI concludes the paper.

II. KinectFusion Algorithm

Figure 1 outlines the functionality of the KinectFusion algorithm.

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

The **bilateral filter** is a stencil-based, edge-preserving filter that blurs the depth image in order to reduce the effects of noise and invalid depth values [4]. This kernel uses a 5x5

978-3-9819263-5-4/DATE21/©2021 EDAA

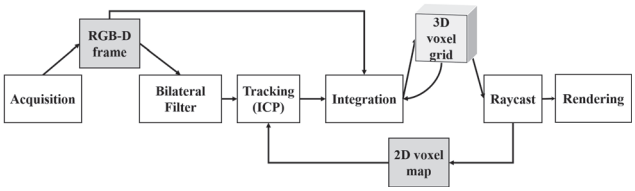

Fig. 1. KinectFusion data processing pipeline [3].

coefficients array, which combines position-based with range filtering. Range filtering averages image values with weights that decay as image dissimilarity increases.

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 [5]. A **reduction** step sums up all the distances of corresponding vertices of the two maps to compute the best match. In the case of a successful match, the input frame is marked as tracked. 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. The three images are formed by recursively sub-sampling the filtered input depth image.

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 [6]. TSDF values are positive in front of the surface and negative behind the surface.

Raycasting is 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).

Figure 2 shows the input and output of the SLAM algorithm for a scene. Figure 3 shows the contribution (%) of each kernel to the total execution time, when running the C++/OpenMP KinectFusion implementations on an ARM Cortex-A53 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.

III. FPGA Architectures and Optimizations

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

Fig. 2. The scene (top left), the input depth (RGB-D) frame (bot. left), the tracking output (top right), and the reconstructed 3D map (bot. right).

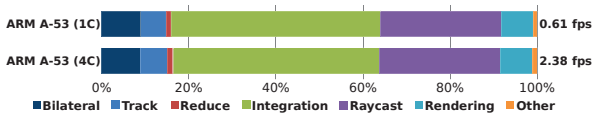

Fig. 3. Contribution of each KinectFusion kernel to total execution time (C=#cores). The y-axis to the right shows throughput in frames/sec.

We experiment with two classes of code optimizations (Table I): (i) **precise** optimizations, which retain the accuracy of the baseline code; (ii) **approximate** optimizations, which may affect the accuracy of the baseline code. 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.

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. Moreover, multiple instances of the same accelerator are used to process different parts of the input frame concurrently to further increase performance. In the following paragraphs, we only describe kernel-specific precise and approximate optimizations.

A. Bilateral Filter

Precise optimizations. The input depth frame is padded 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.

Approximate optimizations. (i) Use a smaller 3x3 coefficient array (instead of 5x5), (ii) use half precision 16-bit floating point format (fp16) instead of 32-bit float, and (iii) remove the range filtering stage of the bilateral filter (which eliminates the invocation of an exponent function in the coefficient filter).

B. Tracking

The tracking kernel accesses the depth image row-wise and pairs each input pixel to its corresponding point in the 2D projection of the reconstructed model. Tracking is invoked multiple times for each frame. For example, the maximum number of iterations for the higher-resolution frame at level 0 of the pyramid is 10.

Approximate optimizations. (i) Loop perforation is an approximate computing technique used to skip loop iterations [7]. We employ this technique to skip pixels of the input depth

TABLE I
PRECISE (TOP ROWS) AND APPROXIMATE (BOTTOM ROWS) OPTIMIZATIONS FOR THE KINECTFUSION KERNELS.

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

frame. (ii) We use half precision fp16 format for all FP operations. (iii) We skip the invocation of the tracking kernel for one or more levels of the pyramid, and reduce the maximum number of iterations at each level of the pyramid.

C. Integration

This kernel consists of a triple nested loop which updates the 3D voxel grid (256x256x256 voxels) using the new pose of the agent obtained by tracking.

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, 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. (ii) We use half precision fp16 format. (iii) We eliminate some of the branches that are used in the loops to check for special conditions. (iv) We replace expensive floating point operations which have low variation across loop iterations with simpler functions or constant values.

D. 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. All approximate optimizations shown in Table I refer to a software implementation of raycasting.

Approximate optimizations. (i) Ray traversal uses steps of variable size. The step starts with a larger size, which becomes smaller as the ray approaches a surface or the edges of the 3D 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 perforation) (iii) Raycast trilinear interpolation accesses 8 different TSDF values for averaging. We provide the option to use simpler interpolation schemes that require fewer TSDF accesses and/or FP operations. (iv) We use fast math to approximate expensive floating point operations. (v) We perform raycast at a lower frequency, once every two frames.

IV. EXPERIMENTAL EVALUATION

A. 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. The FPGA fabric runs at 300 MHz in all our experiments. We measure power dissipation on the FPGA using the Power Management Bus (PMBus) protocol which monitors multiple power rails.

As input, we use three camera trajectories lr.kt[0-2] from ICL-NUIM, a synthetic dataset providing RGB-D sequences from a living room model [8]. Trajectory lr.kt3 is typically excluded from evaluation due to the large number of untracked frames even in precise computation. We run all frames of each trajectory, lr.kt0:1510, lr.kt1:967, lr.kt2:882, but, for brevity, we report results from lr.kt2. In all experiments, we use default SLAMBench parameters as follows: (i) input depth frame resolution is 320x240. (ii) Integration rate is once per frame. (iii) Number of pyramid levels in tracking is 3. (iv) Volume resolution for the scene reconstruction is 256^3 voxels. (v) Volume map (dimensions of the reconstructed scene) is $4.8^3 m^3$.

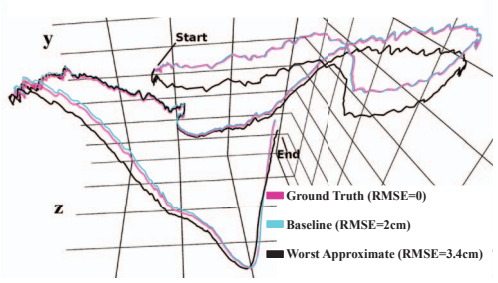

Fig. 4. Trajectories with various average RMSE values. Grid unit distances are 0.5m in x-axis, 0.05m in y-axis, and 0.5m in z-axis.

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. Figure 4 illustrates 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 than even the baseline KinectFusion is an approximation of the ground truth. To place an upper bound on errors, we discard all approximate configurations that increase the number of untracked frames wrt. the baseline. We have found experimentally that an average RMSE higher than 3.4cm (for lr.kt2) almost always results into more untracked frames.

B. Results

Design Space Exploration. Figure 5 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. For comparison, we also include the fastest implementation on ARM (without exploiting the FPGA), using 1 and 4 threads (with both static and dynamic OpenMP scheduling). Note how aggressive approximations increase throughput substantially without a large RMSE.

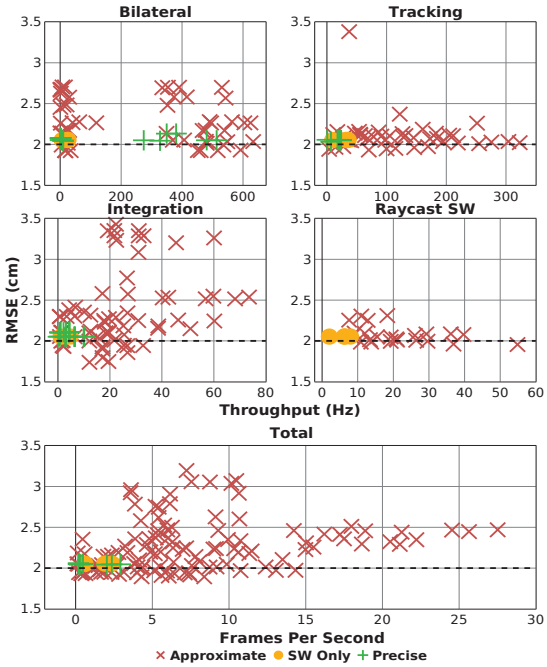

Fig. 5. The scatter plots show throughput vs. average RMSE of various SW and HW kernel implementations when running the *lr.kt2* benchmark. Each red mark corresponds to a valid precise or approximate HW configuration using a combination of the optimizations shown in Table I. Precise HW in the *Total* plot corresponds to a configuration in which all kernels are executed precisely. In *Total*, raycast is executed in the ARM CPU and all other kernels in HW. Baseline average RMSE is 2cm (horizontal dotted line).

Table II shows the performance improvements achieved by the fastest precise and approximate accelerators when all optimizations of Table I are employed (for N=1). 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 II). For the integration kernel, the speedups are 14x and 59x, respectively. These two kernels are characterized by regular memory access 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. *Int_NCU* is also beneficial and provides speedup almost linear to the number of accelerators N. For the tracking kernel, skipping the two higher pyramid levels and/or reducing the maximum iterations in each level has a very beneficial effect on performance.

TABLE II
PERFORMANCE OF HW KERNEL IMPLEMENTATIONS (1 ACCELERATOR).

	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
Raycast (HW)	0.22	0.28 (1.27x)	5.7 (25.9x)	2.22

As we mentioned in section III-D, the raycast kernel is not suitable for hardware acceleration mainly due to complex memory accesses in the large 64 MB 3D voxel grid. Given that the input of the raycast kernel is the voxel grid which has been updated by the integration kernel, we have also evaluated the possibility of merging these two kernels in a single accelerator. The objective is to increase memory access locality by having the raycast kernel immediately consume the updated TSDF values produced by integration. One of the issues with this approach is that the integration kernel does not update the TSDF values in the same order as the one used by the raycast kernel, requiring to store a large number of partial values. Even worse, the need to perform 8-voxel interpolation for each ray element produces multiple out-of-sequence accesses. For these reasons, the raycast kernel is executed on the ARM CPU in our final MPSoC implementation (*Total* in Fig. 5).

Significance Analysis. To better quantify the significance of individual optimizations of Table I on performance, we use Lasso [9], 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.

Table III shows the features with the largest coefficients, 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

TABLE III
LASSO ANALYSIS OF KINECTFUSION THROUGHPUT

Bilateral Filter (MSE=0.039)		**Tracking** (MSE=0.077)		**Integration** (MSE=0.0008)		**Raycast** (MSE=0.0016)		**Combined** (MSE=0.0048)	
Feature	Coeff.	Feature	Coeff.	Feature	Coeff.	Feature	Coeff.	Feature	Coeff.
BF_Unroll^2	0.126	Tr_Pipe^2	0.078	Int_Pipe^2	0.184	R_LP	-0.145	Int_Pipe^2	0.085
BF_Pipe*BF_Unroll	0.099	Tr_Pipe*Tr_LP	0.062	Int_Pipe*Int_Inter	-0.167	R_Rate	-0.110	Int_Inter*Int_Pipe	-0.056
BF_Pipe	-0.095	Tr_LP	-0.039	Int_Inter^2	0.156	R_Step	-0.057	Tr_Pipe^2	0.044
BF_Unroll*BF_Coeff	0.064	Tr_Pipe*Tr_LvlIter	0.031	Int_SLP	-0.120	R_Fast	-0.056	Int_Pipe*Int_Unroll	-0.043
BF_Coeff	-0.060	Tr_LvlIter	0.021	Int_Inter*Int_SLP	0.113	R_LP*R_Rate	0.049	Int_Inter*Int_Unroll	0.039
		Tr_Pipe	-0.003	Int_Unroll	-0.039	R_TrInt	-0.047	BF_Unroll^2	0.037

TABLE IV
LASSO ANALYSIS OF KINECTFUSION OUTPUT RMSE

Bilateral Filter (MSE=0.019)		**Tracking** (MSE=0.008)		**Integration** (MSE=0.017)		**Raycast** (MSE=0.038)		**Combined** (MSE=0.0052)	
Feature	Coeff.	Feature	Coeff.	Feature	Coeff.	Feature	Coeff.	Feature	Coeff.
BF_Coeff^2	0.408	Tr_LP	-0.091	Int_SLP*Int_CLP	-0.255	R_LP^2	-0.166	R_Step	-0.035
BF_Range	-0.237	Tr_LvlIter	-0.060	Int_CLP^2	-0.219	R_TrInt^2	0.166	Int_SLP	0.032
BF_Coeff*BF_Range	0.056	Tr_LvlIter*Tr_NCU	0.046	Int_FPOp^2	0.169	R_LP	0.127	Tr_LvlIter*R_LP	-0.027
BF_Coeff*BF_HP	-0.048	$Tr\_LvlIter^2$	0.043	Int_SLP	-0.083	R_LP*R_TrInt	0.096	Int_SLP*Tr_LvlIter	-0.025

indicates non-linear relations between optimizations and performance.

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.

Table V 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 *Combined* column of Table V to achieve 11.2x speedup. Then, *Conf2* incrementally includes all 15 optimizations shown for each individual kernel of Table III. When the first 22 (out of 27) optimizations of the Lasso ranking are enabled, the entire application achieves top performance at 27.5 fps and RMSE=2.47cm. The 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.

TABLE V
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

The fastest approximate configuration consists of one bilateral, one tracking and four integration kernels (all approximate), and has average RMSE equal to 2.5cm. 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. 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. 1), we can only overlap the bilateral filter, gaining 4 ms per frame.

Table IV presents a similar Lasso analysis to quantify the effects on the average RMSE due to the approximations applied on each kernel separately and on all kernels in combination. Typically, loop perforation, if not applied judiciously, has the most negative effect on output error.

Power and Area exploration. The average power of the PL fabric 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.

Figure 6 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.

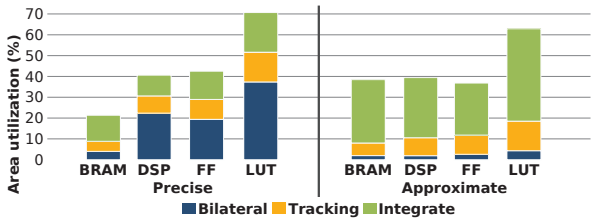

Fig. 6. Area utilization for the fastest precise and approximate configurations.

Timeline analysis. The timelines of Figure 7 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 in lr.kt1.

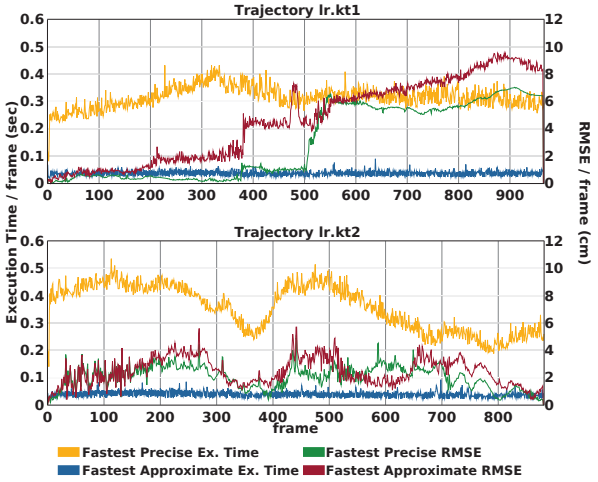

Fig. 7. Timeline showing execution time and RMSE per frame for the fastest precise and approximate FPGA implementations.

V. Related Work

It is only recently that dense SLAM algorithms have been ported on FPGAs. In [10], an extensive design space exploration is performed for the InfiniTAM algorithm [11] (which is derived from KinectFusion [12]), 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. 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 [10] result in unacceptably high error and number of untracked frames.

ORB-SLAM is a feature-based sparse SLAM that operates in real time without acceleration in indoor and outdoor environments [13]. A hardware implementation of ORB-SLAM is presented in [14]. Abouzahir et al. evaluate a number of sparse SLAM algorithms in desktop and embedded platforms [15].

Semi-dense SLAM algorithms such as LSD-SLAM [16] provide a more dense and information-rich representation compared to sparse methods, while achieving higher computational efficiency from processing a subset of high quality observations. In [17], LSD-SLAM is accelerated on an FPGA SoC, achieving more than 60 fps on a 640x480 input frame. Previous work by the same authors describes an FPGA accelerator only for semi-dense tracking on embedded platforms [18]. A very low-power ASIC design for real-time visual inertial odometry (VIO) targeting nano-drones has been announced recently [19]. Approximate computing has been used to accelerate SLAM implementations. In [20], [21], the degree of approximation is dynamically adjusted during the motion of the robot.

VI. Conclusions

In this paper, 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 achieves 27.5 fps at an 320x240 input depth frame resolution without exceeding the tight error bounds necessary for tracking convergence.

Acknowledgement

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 code: T1EDK-01149).

References

[1] https://github.com/csl-uth/KinectFusion-fpga.
[2] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, 2016.
[3] B. Bodin *et al.*, "SLAMBench2: Multi-Objective Head-to-Head Benchmarking for Visual SLAM." in *ICRA*, 2018.
[4] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images," in *ICCV, Bombay, India, Jan.*, 1998.
[5] P. J. Besl *et al.*, "A Method for Registration of 3-D Shapes." *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, 1992.
[6] B. Curless and M. Levoy, "A Volumetric Method for Building Complex Models from Range Images." in *SIGGRAPH*, 1996.
[7] S. Sidiroglou-Douskos *et al.*, "Managing Performance vs. Accuracy Trade-Offs with Loop Perforation," in *ESEC/FSE, Szeged, Hungary, Sept.*, 2011.
[8] A. Handa *et al.*, "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM," in *ICRA Hong Kong, China, May*, 2014.
[9] R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society (Series B)*, vol. 58, 1996.
[10] Q. Gautier *et al.*, "FPGA Architectures for Real-time Dense SLAM." in *ASAP*, Jul 2019.
[11] O. Kähler *et al.*, "Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices," *IEEE Trans. Vis. Comput. Graph.*, vol. 21, no. 11, 2015.
[12] R. A. Newcombe *et al.*, "KinectFusion: Real-time dense surface mapping and tracking," in *ISMAR*, 2011.
[13] R. Mur-Artal *et al.*, "ORB-SLAM: A Versatile and Accurate Monocular SLAM System," *IEEE Trans. Robotics*, vol. 31, no. 5, 2015.
[14] W. Fang *et al.*, "FPGA-based ORB Feature Extraction for Real-Time Visual SLAM." *CoRR*, vol. abs/1710.07312, 2017.
[15] M. Abouzahir *et al.*, "Embedding SLAM Algorithms: Has it come of age?" *Robotics and Autonomous Systems*, vol. 100, 2018.
[16] J. Engel *et al.*, "LSD-SLAM: Large-Scale Direct Monocular SLAM," in *ECCV-Part II*, 2014.
[17] K. Boikos and C. S. Bouganis, "A Scalable FPGA-Based Architecture for Depth Estimation in SLAM," in *ARC*, 2019.
[18] ——, "A High-Performance System-on-Chip Architecture for Direct Tracking for SLAM," in *FPL*, 2017.
[19] A. 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*, vol. 54, no. 4, 2019.
[20] Y. Pei *et al.*, "SLAMBooster: An Application-Aware Online Controller for Approximation in Dense SLAM," in *PACT*, 2019.
[21] ——, "A Methodology for Principled Approximation in Visual SLAM," in *PACT*, 2020, pp. 373–386.