High-Level Synthesis for FPGA Design Based-SLAM Application

Mohamed ABOUZAHIR, Abdelhafid ELOUARDI and Samir BOUAZIZ
SATIE-CNRS UMR 8029, Paris-Sud University-Paris Saclay
University Digiteo Labs
91405 Orsay Cedex, France
email{*first name.last name@u-psud.fr*}

Omar HAMMAMI and Ismail ALI
Groupe de Recherche Ingenierie Système
ENSTA ParisTech, 828 Boulevard des Maréchaux
91120 Palaiseau, France
email{*first name.last name@ensta-paristech.fr*}

***Abstract*—The development of SLAM algorithms in the era of autonomous navigation and the growing demand for autonomous robot in place of human being, has put into question how to reduce the computational complexity and make use of these algorithms to operate in real time. Our work aims to take advantage of the high level synthesis (HLS) on FPGAs to design a real time SLAM application. Precisely, we evaluate the promess-held by the new modern low power FPGAs in accelerating SLAM algorithms. Throughtout this, we will attempt to implement a well-known algorithms (FastSLAM2.0), on a new modern FPGA using OpenCL, a standard high level language. Our implementation results show a significant improvement of the algorithm processing time on an FPGA device over a modern powerful embedded GPGPU.**

***Keywords*—Embedded Architectures, High Level Synthesis, OpenCL, FPGA, SLAM.**

I. Introduction

It is often desired in computation-intensive applications to take some of the load off the CPU, by letting a piece of hardware perform some intensive tasks. GPUs are commonly used for this purpose, but they impose strict limitations on the accelerated algorithm, that must be met in order to achieve an effective speedup. Field Programmable Gate Arrays (FPGAs) present an attractive alternative in many cases, as the accelerating hardware doesn't have a hardwired architecture of its own. Rather, the algorithm under test shapes the architecture; logic building blocks are placed in parallel or pipelined to achieve a high utilization of the device's capacity, depending on the expected data flow. FPGA has always been considered as the generation of integrated circuit which is able to be fully reconfigured by user. An FPGA is commonly claimed to perform higher performance with reliability. Although, when employing traditional logic design techniques, the human effort necessary to harness the FPGAs capabilities for a complex algorithms often makes the FPGA an unattractive choice.

High-Level Synthesis (HLS) [1] is a recent technique for utilizing programmable logic without using the traditional hardware definition languages (Verilog / VHDL). The HLS converts a high level language into a hardware description, aiming to utilize the programmable device efficiently for accelerated operations and economic resource usage. This opens an opportunity for regular programmers to design custom co-processing architectures in efficient way.

The OpenCL [2] standard is a high-level and unified programming model for accelerating algorithms on heterogeneous systems. OpenCL allows the use of a C-based programming language for developing code across different platforms, including FPGAs. The arrival of OpenCL for FPGAs gives us an opportunity to usher in the era of heterogeneous computation for embedded devices as well. It allows a user to abstract away the traditional hardware development flow for a much faster and higher level software development flow for complex algorithms.

SLAM (Simultaneous Localization And Mapping) [3] algorithms are computationally intensive. Therefore, there is a general need, in case of embedded systems, to have an architecture that allows a software optimization for efficient and scalable implementation. The FPGAs [4] have an advantage over an ordinary processing unit. An FPGA does not waste compute cycles doing unnecessary processing. In other words, an FPGA excels for doing one simple and repetitive processing task. This great advantage of FPGAs has catalyzed investigations on its potential usage for optimizing the SLAM algorithms.

Our contributions: The arrival of high-level synthesis gives us an opportunity to design a hardware implementation of complex algorithms such as SLAM algorithms as well. However, the question remains open whether and what are the optimization that must be performed to allow an efficient and scalable implementation of a heavy computationally algorithms on an FPGA given the limited ressources. Towards this, we implement a well known SLAM algorithm deployed in numerous potential applications for autonomous robot navigation on an FPGA. Our work aims also to investigate wether a dedicated architecture is suitable to design real-time SLAM systems. To the best of our knowledge, this is the first paper presenting a full implementation of such algorithm on an FPGA and gives a comparative study against an embedded GPGPU. Previous works, were almost exclusively concerned about implementing only some parts of the SLAM algorithm using hardware description languages.

The paper is organized as follows: In section II, we give an

978-1-5090-4320-0/16/$31.00 ©2016 IEEE

overview about the state of the art and motivation behind our work. Section III gives an overview about the target SLAM algorithm as well as the parallel OpenCL implementation. In Section IV, we describe the methodology for evaluating the SLAM algorithm, we give also a description about the platform and hardware used in evaluation. In Section V, we describe the SLAM algorithms implementation results as well as the experimental results on running times, individually, for each major kernel function. We compare the result against an embedded GPGPU. Last section provides a more holistic viewpoint of the results and conclude the paper.

II. Related Work and Motivation

Major strides have been made in SLAM algorithms computation over the last years. Almost all threads of previous work, singularly focused on improving SLAM performances without discussing other issues that arise specially in embedded low power architectures. Hence, the existing body of work dose not provide any insight into opportunities and challenges to tackle with embedded SLAM algorithms. Of late, many initial attempts have been made to bridge this gap. However, almost all of them mainly do note take into account the issue of algorithm architecture matching. Actually, a SLAM system consists to use a recent embedded architecture as powerful as possible with a suitable SLAM algorithm, and then try to optimize the algorithm as maximum as possible.

In recent paper, [5] implemented an accelerator block of EKF-SLAM for 3D visual SLAM. They implemented the block on a low power embedded architecture integrating a processor with reconfigurable hardware. The sole overarching aim of their work is to optimize the algorithm on a dedicated architecture. They did not take into account the SLAM system in itself. In essence, the EKF algorithm is not a suitable choice for SLAM problem since the complexity scales linearly with the number of landmarks and hence the EKF is not suitable for use in large scale environment. Furthermore, the EKF SLAM is not suitable to take advantage of the massively parallel architecture of the FPGA, as most part of the algorithm are sequential.

[6] proposed an hybride architecture to execute an EKF-SLAM. Landmarks detector is ac- celerated using a dedicated hardware architecture while the heart of the algorithm is implemented on a Microblaze processor. Performances obtained by the system are promising. At 30 Hz, the Kalman filter can contain up to 30 landmarks and observe up to 6 landmarks. However, such SLAM system suffer from complexity and can not be used in real time in outdoor/indoor environments.

Authors in [7] implemented a parti- cle filter based SLAM on an FPGA. They implemented a robust laser FastSLAM1.0 with odometry sensors. They real dataset to evaluate the FPGA implementation. Most part of the FastSLAM1.0 is implemented on the Microb- laze soft-core processor including the sampling, impor- tance weight and resampling blocs. Other parts are hard- ware accelerated on FPGA. Their work suffers from a set of different issues. The implementation is not fully hardware accelerated where some parts of the algorithm turns on the soft core. The highly parallel architecture of the FPGA is not fully exploited. Moreover, they im- plemented the Fast-SLAM1.0 algorithm which suffer from samples impoverishment issues and particles depletion [8]. Thus, such algorithm is not a good choice when dealing with real outdoor/indoor large scale environment.

In contrast, the aim of this paper is to develop a real-time application of FastSLAM 2.0 using a new acceleration technologies based on modern heterogeneous architectures

III. Algorithm Implementation and Functional Block Partitioning

A. FastSLAM2.0 Overview

SLAM algorithms allow autonomous navigation of robots in unknown environments. Localization and mapping represent a concurrent problem that cannot be solved independently. Before the robot can estimate the position of a given landmark, it needs to know from which location this landmark was observed. At the same time, it is difficult to estimate the actual position of the robot without a map. A good map is necessary for localization while an accurate pose estimate is needed for map reconstruction. The FastSLAM2.0 [8] addresses an issue of the SLAM problem and allows a robot to navigate in an unknown environment. Such algorithms is deployed in numerous potential robotic applications and autonomous navigation systems. Throughout this work, we are interesting in a high level synthesis of this algorithm on a target FPGA. A SLAM algorithm relies on sensors data to concurrently estimate both map and robot pose. Two sensors are usually used: proprioceptive and exteroceptive sensors [9]. Throughout this work we used a monocular camera as an exteroceptive sensor to observe environment and odometers as proprioceptive sensors to estimate robot pose. Algorithm 1 describes the monocular FastSLAM2.0

1) Prediction: The prediction consists of sampling a new robot pose at each timestamp given each particle. It calculates the new particle pose by incorporating the odometers data [10]. Thus, this task calculates the future state of the particles pose and their initial covariance matrix to be used in the particle update task.

2) FAST Detection and Matching: The detection phase consists of extracting features from an image using FAST detector [11]. Once features are extracted, a matching task is then performed. The matching task projects map landmarks into the current frame and perform a matching between observation and landmarks using a similarity based-correlation metric.

3) Update particle state: This task ameliorates particles pose and their uncertainty to maintain diversity in particles and to prevent sample impoverishment. The update process is performed incrementally for each particle according to the number of matched landmarks [12].

4) Map Estimation: If a landmark has been previously observed, its position in the image is predicted using the pinhole model and its position is updated using EKF (Extended

Algorithm 1 Visual FastSLAM2.0

Particles Initialization
while 1 **do**
  $u_t \leftarrow (n_l, n_r)$ ▷ Odometric data acquisition
  PREDICTION :
  for Each Particle **do**
    $s_t^m = f(s_{t-1}^m, u_t)$ ▷ Predicte new particle pose
  end for
  IMAGE PROCESSING :
  $(u_k, v_k) \leftarrow$Feature Extraction
  select high particle weight s_t
  for Each Landmark **do**
    $(\hat{u_n}, \hat{v_n}) \leftarrow h(s_t, X_n)$ ▷ Pin-Hole model
    $zmssd((\hat{u_n}, \hat{v_n}), (u_k, v_k))$ ▷ Select landmark with small zssd
  end for
  PARTICLE UPDATE :
  for Each Particle **do**
    $s_t^m \sim N(\mu_n^m, \Sigma_n^m)$ ▷ Sample new particle pose
  end for
  ESTIMATION :
  for Each Particle **do**
    for Each Landmark **do** $(X, C)_n^m \leftarrow kalman(X_n^m, C_n^m, \hat{z_n}, z_n)$ compute weight ω^m
    end for
  end for
  INITIALIZATION :
  for Each Particle **do** Compute $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$ ▷ Inverse depth parametrization
  end for
  RESAMPLING : importance resampling
end while

Kalman Filter). The weight of each particle in the filter is then computed according to this landmark.

5) Map Initialization: This task adds the observed landmarks that are not matched in the estimation phase to the map. For each new landmark its calculates the initial state, the covariance matrix and adds it to each particle map.

6) Particle Resampling: This task deletes improbable particles with a low weight and duplicates particles that received a high weight. A new set of particles is selected with replacement probabilities in proportion to the weights. This sampling technique is called the importance resampling.

B. Functional Block Partitioning

A SLAM algorithm does not have a fixed processing time which can be expected beforehand, because of dependencies on many parameters. In order to evaluate the processing time, the algorithm was analyzed in terms of instruction and operations order. This study allowed us to divide the algorithm into independent Functional blocks (FBs) with a bounded processing time Fig.1.

- **FB1** Prediction: the prediction task propagates the current state of the particles in the filter via a probabilistic motion model [10]. It calculates the future state by incorporating the odometric data. The operations are independent for each particle and hence the FPGA can process each particle independently. To parallelize this block, we transfer to the FPGA both the current state of the particle and the encoders data $u(n_r, n_l)$. The motion model must be applied with a randomization to reflect the random error of the system and the noise of sensors. The random numbers are generated by the host and then transferred to the FPGA.
- **FB2** Image processing: this block extracts features from an image using FAST detector. The processing time of this block depends on the image size. The matching tasks projects map landmarks into the current frame, and performs a matching task between observation and landmarks using ZMSSD (Zero-Mean Sum if Squared Distance). The matching is then performed according to the particle that has the high importance weight [13]. So the processing time is not constant and depends on the number of landmarks to be projected, and the number of current observations.
- **FB3** Particle Update: the FastSLAM 2.0 is a modified version of the particle filter to deal with sample impoverishment issue. In essence, to prevent the resampling step from being more wasteful with the particles, we compute a new Gaussian mean and covariance based on the most recent observation. Then, a new particle pose is sampled from the new proposal distribution. This requires the development of a new procedure of sampling a pose from the new proposal distribution. This functional block can be parallelized on the FPGA since each Gaussian can be constructed independently. We transfer to the FPGA for each particle: the initial covariance matrix P_t^m, the particle pose s_t^m, the parameters of the landmarks in the map that were matched $(x_0, y_0, \rho, \theta, \varphi, C_t)$ and their corresponding observation in the current frame (u, v). The new proposal distribution is constructed incrementally for each particle according to each matched landmark.

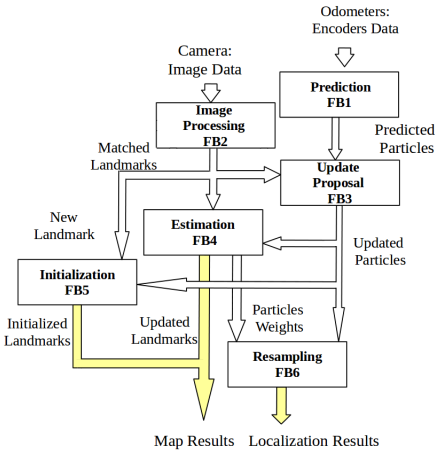

Figure 1. FastSLAM2.0 FBs partitioning

- **FB4** Estimation: during the estimation step, if a landmark in the map is matched with a current observation, its position in the image $(\hat{u},\hat{v})$ is predicted using the pin-hole model [14]. The innovation between the predicted pose and the observation (u, v) is computed. The extended Kalman Filter (EKF) corrects the inverse depth parametrization (ρ, θ, ϕ) of the matched landmark and its covariance matrix. The likelihood to get the sensors readings from these particles hypothesis is computed according to the observation. Since the fact that each particle has its own map, the operation mentioned above can be done independently and hence the estimation functional block can be parallelized on the FPGA. For each particle, we transfer to FPGA : the particle pose s_t^m and the landmark parameters $(x_0, y_0, \rho, \theta, \varphi, C_t)$ to predict its position in the image, the observation (u, v) for innovation computation and P_t^m to compute the new importance weight.
- **FB5** Initialization: in our implementation we used the inverse depth parametrization for landmark initialization [15]. The initial coordinates of the landmark are $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$, where x_i and y_i correspond to the first view camera pose, θ_i is the azimuth, φ_i is the elevation and ρ_i is the inverse depth.
- **FB6** Resampling: to prevent the particles depletion, the resampling functional block replaces unlikely particles with likelier ones. A new set of trajectories is sampled proportionally to the corresponding weights. The most time-consuming parts in the resampling step are accelerated on the hardware. We note that this block has two main parts: Resampling particle state (processed on FPGA) and Resampling particle map (processed on CPU since the map is arranged in a binary tree on CPU). One of the bottle-necks in this block is the total weight calculation and the resampling threshold N_{eff} ($N_{eff} = 1/\left(\sum_{i=1}^{M} \omega_i^2\right)$), when ω_i is the normalized weight).

C. OpenCL Algorithm implementation

1) Image Processing (FB2): The image processing task uses the FAST detector. It is a sequential algorithm already optimized using machine learning. So it is implemented on one-core of the CPU. The matching between landmarks and observations is performed according to one particle that has the highest weight. So it is parallelized on the quad-core CPU according to the observed landmarks.

2) FB1 OpenCL implementation: The particle state is initialized and saved in a buffer in the GPU memory. To reflect the odometric data noise, we suppose that it is modeled by a Gaussian following the formula:

$$n_l = n_l + 4 * \left(\sqrt{(-2.0 * \log(\epsilon_1))} * cos(2 * \pi * \epsilon_2)\right) * \epsilon_g \quad (1)$$

$$n_r = n_r + 4 * \left(\sqrt{(-2.0 * \log(\epsilon_3))} * cos(2 * \pi * \epsilon_4)\right) * \epsilon_g \quad (2)$$

where n_l, n_r are the left and right wheels encoders data.$(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4)$ are random numbers between $[0, 1]$, and ϵ_g is the slipping noise since the wheels may slip and generate a gap between measurement and the real traveled distance. ϵ_g may have a small value if the robot is moving with a low velocity, which is the case for the Rawseeds robot [16]. To implement the randomization procedure on FPGA using OpenCL, we note that each particle is processed by one thread, each thread has its individual identification. Therefore, instead of generating the random numbers $(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4)$ for each particle, we just need to build a set of random numbers with limited quantity on the CPU (60 values in our implementation) and transfer them to the FPGA. Then, we combine this set using identification to have the private random value for each particle. Once the noise is computed within the kernel, the particle pose is computed and stored in the same memory buffer.

3) FB3 OpenCL implementation: In the OpenCL implementation, we transfer the matched landmarks parameters and the state of all particles in once. Therefore, this allows the Gaussian construction to be done within the kernel. The resulting Gaussian is saved in the global memory since it is a read/write memory. Gaussian construction is done inside the kernel. In essence, one kernel execution completely construct the Gaussian because all the matched landmarks are transferred to global memory before the kernel execution. Therefore, the OpenCL implementation is efficient. It allows the Gaussian construction for all particles to be done in one kernel execution. This reduces data transfer at each iteration implying a significant improvement.

4) FB4 OpenCL implementation: The implementation of the estimation functional block in OpenCL is quite faster. We recall that in the FB2 OpenCL implementation we already transferred all the matched landmarks parameters to global memory. Therefore there is no transfer that comes before and after the kernel execution. Thus, we just need to directly activate the kernel that implements the general EKF equation to update all the matched landmarks parameters within the kernel in one execution. As a result, the implementation of the estimation functional block in OpenCL allows a significant improvement.

5) FB5 OpenCL implementation: The implementation of this block in OpenCL is quite different. When transferring the matched landmarks parameters in the particle update (FB2), we transfer also their corresponding observation parameters. So the kernel is executed without any previous transfer. It computes the inverse depth parametrization for all the new landmarks within the kernels and this is done for all particles. Note that we use the binary tree for map management as mentioned earlier. Therefore, all the matched landmarks that have been updated (FB3) and initialized (FB4), are transferred back in this step to the host CPU.

6) FB6 OpenCL implementation: To implement the weight summation using OpenCL, we divide the sum into K sub-tasks. Each sub-task takes the sum of all the particles weights which have the same surplus. The corresponding identification

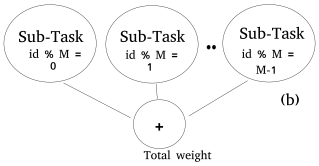

Figure 2. In OpenCL, a group of sub-tasks classified by the remainder (id \% K), computes the weights summation of a chunk of particles. The result is transferred back to the CPU to compute the final total weights.

is divided by K. The total weight is then the sum of all the sub-task results. Each sub-task is processed by one kernel. Tasks can be executed in parallel. Results of sub-tasks are transferred to host CPU to calculate the total weight. The number of sub-tasks K must be a multiple of M (number of particles) . This is illustrated in Fig.2. The computation of the minimum number of effective particles before resampling N_{eff} is also implemented in the same way: instead of computing the sum, we compute the square sum. In OpenCL, the sub-tasks compute the square sum of a chunk of particles weights, and then the final value is computed by the CPU.

IV. Methodology

A. OpenCL Optimization Work Flow

Fig.3 illustrates the OpenCL optimization work flow. By emulating the kernel regarding the specific board that is used (Arria 10 in our work), we can get first detailed statistics about the resulting design, major problems and bottlenecks. The overall optimization process stands as follow:

- The intermediate compilation step checks for syntactic errors. It then generates a **.aoco** file without building the hardware configuration file. The estimated resource usage summary generated can provide insight into the type of kernel optimization that can be performed.
- The OpenCL kernel that target the FPGA can be emulated on one or multiple emulation devices. This allows to asses the functionality of the kernel before targeting the real FPGA hardware.
- The hardware configuration file **.aocx** contains the Verilog description of the OpenCL kernel. During execution, we can collect performance information and get them reviewed in a Profiling GUI.
- After optimization, a full compilation can be performed and the generated **.aocx** file can be executed on the hardware.

B. Platform Description

The host computer used in our evaluation operates at 2.5 GHz under Red Hat Enterprise Linux 7.2 with a 64 GB RAM (random-access memory). The development in this paper was done using specific tools dedicated to Altera devices design which are not available to the public realm. We used also a recent BSP (Board Support Package) for OpenCL implementation on the Arria 10 SX660 provided by the board vendor. The SLAM algorithm was developed in OpenCL using the Altera

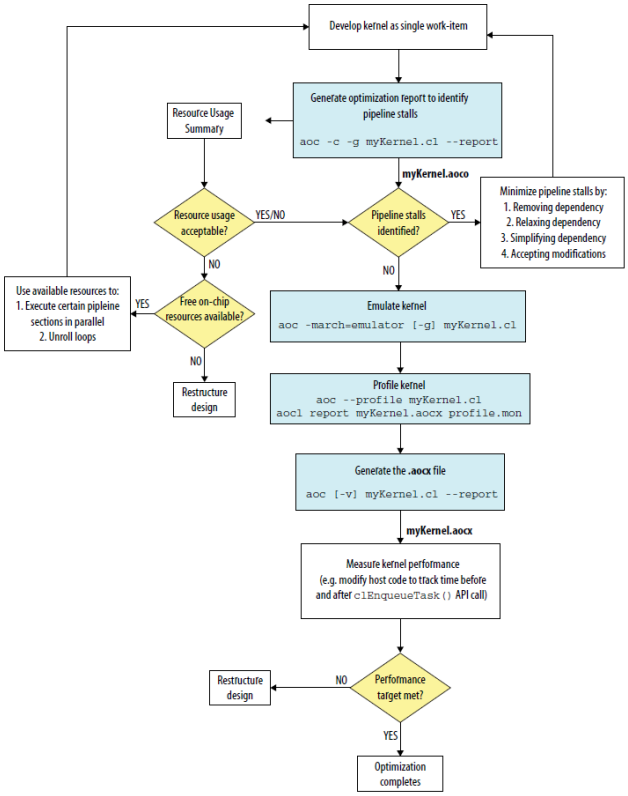

Figure 3. Altera OpenCL kernel compilation work-flow. It contains different stage for optimizing the hardware configuration.

SDK (System Development Kit) for OpenCL. The Altera SDK for OpenCL allows avoiding the traditional hardware FPGA development in order to achieve a much faster and higher level software development flow. It includes multiple optimizations and can produce deep reports of the compilation and the code optimization. Furthermore, we used the Quartus II Prime Pro edition for synthesis, it is the only software that support the Arria 10 device.

For performance evaluation, we have implemented the algorithm on a modern embedded GPGPU. The architecture used in our evaluation is the NVIDIA Tegra $K1$ tab.I. Tegra $K1$ is a recent system on a chip (*SoC*) developed by Nvidia for mobile devices and multimedia applications. K1 processor integrates a quad-core ARM Cortex A15 CPU running at 2.3 GHz and an NVIDIA Kepler GPU with 192 NVIDIA CUDA cores. The embedded GPU adopts a unified-shader architecture. Unified Shading Architecture hardware is composed of an array of computing units which are capable of handling any type of shading tasks instead of dedicated vertex and fragment processor as in old GPUs. The full description of the FastSLAM 2.0 implementation on the Tegra K1 is given in our previous work [17].

C. Description of Arria 10 FPGA

We used the Arria 10 FPGA as a target platform for our SLAM implementation. Arria 10 is one of the latest chip produced by Altera delivering the highest performance at 20

Table I
NVIDIA TEGRA K1 SPECIFICATIONS

GPU	CPU
192 NVIDIA CUDA Cores	Quad-Core ARM Cortex-A15
Clock speed: 852 MHz	Clock speed: 2.3 GHz
OpenGL version: 4.4	OS: Linux for Tegra

Table II
RESOURCES OF ARRIA 10

Resource		Arria 10 device SX 660
Logic Elements (LE) (K)		660K
ALM		251,680
Register		1,006,720
Memory	M20K	42,620
	MLAB	5,788
DSP Blocks		1,687
18 x 19 Multiplier		3,374
17.4 Gbps Transceiver		48
PCIe Hard IP Block		2
Hard Memory Controller		16

nm. Arria 10 FPGA based SoC is a low power embedded architecture up to 40% lower power than previous FPGAs generation. It allows up to 1500 GB/s floating-point operation with DSP blocks. The system clock is 100 MHz. The chip also includes a Dual-Core ARM operating at 1.5 GHz. Fig.II shows the available ressources in terms of logic elements, DSP and memory blocks of the Arria 10 FPGA.

D. Real dataset based evaluation

The SLAM applications are an interesting topic when it comes to explore a real indoor/outdoor environment. Evaluation of SLAM algorithms requires a set of different sensors data which are necessary for benchmarking. Sensors data can be obtained either by using a real instrumented robot or an available dataset. Furthermore, the consistence of SLAM algorithms depends on many parameters of the environment such as the area of the environment and the number of visible landmarks. Thereby, in our evaluation the SLAM algorithm was evaluated using a real indoor dataset [18] [16]. The dataset provides a set of different sensors data. In our evaluation we have used data of encoders as proprioceptive sensors, data of a monocular camera and a laser stream as exteroceptive sensors.

E. Software in the Loop (SIL)

SIL simulation is used to test the real-time execution of the developed algorithm on a prototyped system instead of the actual embedded target [19]. This allows easy debugging of errors thanks to the availability of computing resources and display facilities. Developing the SLAM algorithm with OpenCL using the software in the loop emulation allowed us to abstract away the traditional hardware development flow for a much faster and higher level software development flow. Therefore, we can get a detailed optimization report about the implementation with specific algorithm pipeline dependency information, pushing the longer compilation time to the end when we are satisfied with our kernel performance results.

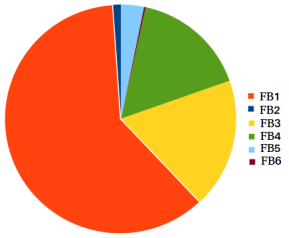

Figure 4. FBs workload on Rawseeds dataset

V. PERFORMANCE EVALUATION

A. Algorithm implementation on an ARM processor using Rawseeds dataset

In practical situations, SLAM applications are interesting topics specially when it comes to explore extremely diffcult situation where the dimension of the robot may not allow using high performance machine to execute SLAM algorithms [20]. Therefore, there is a general need, in case of embedded systems, to have a platform that allows efficient implementation to ensure real-time constraints. In our evaluation, we have implemented the FastSLAM2.0 on one-core embedded ARM processor to identify the processing tasks (FBs) that require considerable amount of time by evaluating their processing times. A parallel implementation of the algorithm on a dual-core ARM was proposed in [21].

Fig.4 presents the workload of each FB when running the one-core implementation with 4096 particles on an embedded ARM processor. FB1 is a time-consuming functional block. It occurs several times in one iteration as many encoders data are acquired to reconstruct robot trajectory using motion model. Also, computing the initial covariance matrix P_m is time consuming. It increases as the number of odometric data increases between two consecutive measurements. FB3 and FB4 occur only when there is at least one matched landmark. Therefore, their complexity depends on the number of matched landmarks. FB5 occurs only when there is a new landmark to add in map. FB6 resamples particles only when there is at least one likelihood computed in FB4 to reflect observation on particles poses.

B. Estimated Resource Usage Summary

Tab.III shows the estimated resources usage of the FPGA when implementing the SLAM OpenCL kernel before and after optimization. The FBs, use 121% of logic elements, 65% of logic registers, 87% of memory blocks and 49 of DSP blocks. A kernel optimization is mandatory in order to accelerate all FBs on FPGA. The second column of Tab.III (after optimisation) shows the estimated resources when optimizing data processing efficiency of the OpenCL kernel. This is done by implementing optimizations strategies such as unrolling loops, setting work-group sizes, specifying number of compute units and work-items and setting optimization options during compilation. All FBs can then be accelerated on the hardware.

Table III
ESTIMATED RESOURCE USAGE SUMMARY (%)

	Before Optimization				After Optimization			
Functional Blocks (FBs)	Logic utilization	Dedicated logic registers	Memory blocks	DSP blocks	Logic utilization	Dedicated logic registers	Memory blocks	DSP blocks
FB1	35	19	22	17	34	19	20	17
FB3	40	20	26	19	33	20	26	19
FB4	29	18	20	13	16	18	20	13
FB5	6	3	6	0	6	3	6	0
FB6	11	5	13	0	8	5	13	0
Total (ms)	121	65	87	49	97	65	85	49

C. Processing Times

Each OpenCL kernel execution consists of three phases: data transfer to device, kernel execution and data transfer back from device to host CPU phase after kernel execution. We would like to note that we have, deliberately overlapped data transfer and kernel execution phases in the time measurement to take into account the overhead of data transfer.

To evaluate the parallel implementation discussed above, FPGA implementations using OpenCL, of the FastSLAM2.0 were run and time-evaluated using Rawseeds datasets gathered in an indoor environment [18]. The evaluation is made using wheels encoders and a monocular camera data. For a comparative study, we implemented the FastSLAM2.0 on an ARM based embedded GPGPU SoC. The GPGPU implementation is done on a modern embedded GPGPU, Tegra K1 with 192 CUDA core. Table IV synthesizes a comparison of processing times of each major kernel function obtained after parallelization when running the algorithm with 4096 particles for 500 iterations.

Let's recall that for the prediction (FB1), we generate random numbers on host CPU and we transfer only 60 random numbers from host CPU to FPGA. This block is executed in $51.66\ ms$ on 192 CUDA Cores and $3.84\ ms$ on the FPGA. In the particle update functional block (FB3), for OpenCL implementations, we transfer the matched landmarks for all particles, the matching index and the observations made from host CPU to FPGA. Indeed, the Gaussian construction using OpenCL is done within the compute kernel. All the matched landmarks are held in the global memory. This increases performance and reduces data transfer at each iteration. This block is executed in $32\ ms$ on GPU and $2.16\ ms$ on the FPGA. The results for the estimation task (FB4) are inline with expectations. In fact, as we saw before there is no transfer neither before nor after the kernel execution in the OpenCL implementation since the matched landmark parameters have been already transferred before in FB3 block. This block is executed in $19.45\ ms$ on GPU and 1.32 on FPGA. In contrast, in the implementation of the inverse depth initialization (FB5), there is no data transfer to kernel before execution. However, we transfer back, from device to host CPU, the entire map particles (the updated and the initialized landmarks) in order to arrange them in the binary tree. This block is executed in $9.65\ ms$ on GPU and $0.65\ ms$ on FPGA. In the last functional block (FB6), the execution time is 2.7 and 0.18 respectively on GPU and FPGA. Processing time of FPGA implementation has been decreased by a factor of 12x compared to GPGPU implementation on 192 CUDA Cores.

Table IV
MEAN OF PROCESSING TIMES

Functional Blocks (FBs)	ARM based GPU SoC		Host based FPGA SoC	
	Quad-Core ARM	GPU	Host Desktop	FPGA
FB2	4.83	-	2.01	
FB1	-	51.66	-	3.48
FB3	-	32.17	-	2.16
FB4	-	19.45	-	1.31
FB5	-	9.65	-	0.65
FB6	-	2.7	-	0.18
Total (ms)	120.46		9.97	

The SLAM algorithm implemented in our work uses a monocular camera (bearing-only sensor) to observe environment. The number of particles in such system is necessary to maintain reasonable estimates of pose and landmark uncertainty as stated in [22]. There remains significant challenges to tackle with FastSLAM2.0 based bearing-only sensor intended to operate in large geographic scales. The number of particles necessary is not yet defined which may increase with the environment complexity. However, a large number of particles is needed to achieve an accurate localization results. Therefore, a naive implementation of FastSLAM2.0 with a high number of particles would increase the localization accuracy but at the expense of robot performance to operate in real time. To achieve a compromise between accurate localization and real time performance, we implemented the algorithm for different numbers of particles ranging from 2^2 to 2^{16}. Note that a low number of particles (2^4) is not enough for accurate localization, not as much as (2^{14} or 2^{16}) are needed. Although, this range allows a better analysis of complexity. Tab.V explores the parallel computing power of the FPGA. For even larger number of particles, the speedup achieved by the FPGA implementation meets the real-time constaints. As a result, even if a large number of particles is needed in a complex environment to achieve accurate map and localization results, the FPGA implementation of the monocular FastSLAM2.0 system will

Table V
MEAN OF PROCESSING TIME: FPGA IMPLEMENTATIONS AFTER 500 TIME-STEPS

	FBs: Hardware Accelerated					
Number of Particles	16	256	1024	4096	16384	65536
FB1	0.34	0.55	1.36	3.48	12.34	47.73
FB3	0.67	1.57	1.97	2.16	2.04	6.16
FB4	0.46	0.93	2.27	1.31	1.21	5.71
FB5	0.54	1.54	2.51	0.65	0.95	1.18
FB6	0.25	0.36	0.40	0.18	0.25	0.36
Total (ms)	2.26	4.95	8.51	7.78	16.79	61.14

be always efficient and can operates in real time constraints.

VI. CONCLUSION

This article proposed a high level synthesis of a widely used SLAM algorithm on FPGA using OpenCL and quantitatively evaluated its execution times. The performance of the resulting implementation on FPGA has been compared to 192 embedded GPGPU CUDA cores. Using a real dataset, the parallel implementation on the FPGA is shown to outperform the embedded GPGPU implementation for all FBs. The OpenCL kernel emulation on FPGA device has certain limitations. The hardware configuration file generated does not include optimization. Therefore, it might execute at a significantly slower speed than what an optimized kernel might achieve. Although, The results obtained on the FPGA in terms of acceleration demonstrate that a dedicated architecture can be designed for a SLAM system to operate in real-time constraints.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011. I

[2] OpenCL Working Group et al. The opencl specification, version 1.2, revision 16, 2011. I

[3] Sebastian Thrun and John J Leonard. Simultaneous localization and mapping. In *Springer handbook of robotics*, pages 871–889. Springer, 2008. I

[4] National instrument, introduction to fpga technology: Top 5 benefits, http://www.ni.com/white-paper/6984/en/. 2012, april 16. I

[5] Daniel Törtei Tertei, Jonathan Piat, and Michel Devy. Fpga design of ekf block accelerator for 3d visual slam. *Computers & Electrical Engineering*, 2016. II

[6] Diego Botero, Aurélien Gonzalez, Michel Devy, et al. Architecture embarquée pour le slam monoculaire. In *Actes de la conférence RFIA 2012*, 2012. II

[7] Biruk G Sileshi, Juan Oliver, R Toledo, Jose Gonçalves, and Pedro Costa. Particle filter slam on fpga: A case study on robot@ factory competition. In *Robot 2015: Second Iberian Robotics Conference*, pages 411–423. Springer, 2016. II

[8] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI. II, III-A

[9] Sebastian Thrun. Probabilistic robotics. *Commun. ACM*, 45(3):52–57, 2002. III-A

[10] E. Seignez, M. Kieffer, A. Lambert, E. Walter, and T. Maurin. Real-time bounded-error state estimation for vehicle tracking. *IEEE Int. Journal of Robotics Research*, 28:34–48, 2009. III-A1, III-B

[11] E. Rosten, R. Porter, and T. Drummond. Faster and better: A machine learning approach to corner detection. *IEEE Trans on Pattern Analysis and Machine Intelligence*, pages 105–119, 2009. III-A2

[12] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. Fastslam2.0 an improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003. IJCAI. III-A3

[13] E. EADE. *Monocular simultaneous localization and mapping*. PhD thesis, 2008. III-B

[14] A.J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *IEEE Int. Conf. on Computer Vision*, pages 1403–1410, 2003. III-B

[15] J.M.M. Montiel, J. Civera, and A.J. Davison. Unified inverse depth parametrization for monocular slam. In *Int. Conf. on Robotics: Science and Systems*, pages 16–19, 2006. III-B

[16] Alessandro Giusti Daniele Marzorati Matteo Matteucci Davide Migliore Davide Rizzi Domenico G. Sorrenti Pierluigi Taddei Simone Ceriani, Giulio Fontana. Rawseeds ground truth collection systems for indoor self-localization and mapping. *Autonomous Robots*, 27(4):353–371, 2009. III-C2, IV-D

[17] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif, and Abdelouahed Tajer. Large-scale monocular fastslam2. 0 acceleration on an embedded heterogeneous architecture. *EURASIP Journal on Advances in Signal Processing*, 2016(1):88, 2016. IV-B

[18] Giulio Fontana Matteo Matteucci Domenico Giorgio Sorrenti Andrea Bonarini, Wolfram Burgard and Juan Domingo Tardos. Rawseeds: Robotics advancement through web-publishing of sensorial and elaborated extensive data sets. In *In proceedings of IROS'06 Workshop on Benchmarks in Robotics Research*, 2006. IV-D, V-C

[19] Wook Hyun Kwon and Seong-Gyu Choi. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 115–119. IEEE, 1999. IV-E

[20] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006. V-A

[21] M. Abouzahir, A. Elouardi, S. Bouaziz, R. LATIF, and A. Tajer. Fastslam 2.0 running on a low-cost embedded architecture. In *IEEE. The 13th International Conference on Control, Automation, Robotics and Vision, ICARCV*, Marina bay Sands, Singapour, 2014. V-A

[22] E. Eade and Tom Drummond. Scalable monocular slam. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 469–476, June 2006. V-C