An FPGA implementation of the SMG-SLAM algorithm

Grigoris Mingas, Emmanouil Tsardoulias, Loukas Petrou

Aristotle University of Thessaloniki, Faculty of Engineering, Department of Electrical and Computer Engineering, Division of Electronics and Computer Engineering, 54006 Thessaloniki, Greece

A R T I C L E   I N F O
Article history:
Available online 22 December 2011

Keywords:
Field programmable gate array
Simultaneous localization and mapping
Genetic algorithm
Scan-matching
Co-design

A B S T R A C T
One of the main tasks of a mobile robot in an unknown environment is to build and update a map of the environment and simultaneously determine its location within this map. This problem is referred to as the simultaneous localization and mapping (SLAM) problem. The article introduces scan-matching genetic SLAM (SMG-SLAM), a novel SLAM algorithm. It is based on a genetic algorithm that uses scan-matching for gene fitness evaluation. The main scope of the article is to present a hardware implementation of SMG-SLAM using a field programmable gate array (FPGA). The architecture of the system is described and it is shown that it is up to 14.83 times faster compared to the software algorithm without significant loss in accuracy. The proposed implementation can be used as part of a larger system, providing efficient SLAM for autonomous robotic applications.

C2011 Elsevier B.V. All rights reserved.

1. Introduction
The problem of SLAM is one of the most fundamental in mobile robotics. It is encountered when a robot is placed in an unknown environment and has to explore the environment and build a map while keeping track of its location at all times. The robot uses sensor measurements (e.g. laser or ultrasonic scanners, cameras) and/or odometry measurements (translation, rotation of the vehicle) to acquire information over its environment and perform localization and mapping. SLAM is used in applications such as planetary and underwater exploration, mine mapping and rescue missions. The problem has been researched intensively during the last 10 years because its solution is considered necessary to create truly autonomous robots.

SLAM can be thought of as a “chicken or egg” problem: An unbiased map is needed for localization while an accurate location estimate is needed to build the map [18]. This is the reason why iterative methods are used to solve the problem. Most researchers have proposed the use of probabilistic methods, such as Extended Kalman Filters [18,19] and Particle Filters [12,18]. These solutions take both sensor and odometry measurements as inputs. An alternative approach is based on scan-matching (i.e. registration) techniques [5,24] that usually rely on range measurements (e.g. from a laser sensor) and thus avoid the significant errors introduced by odometry.

One such algorithm is the SMG-SLAM. It was developed for use in the Pandora mobile robot [2], an autonomous rescue robot that can explore disaster sites and locate trapped individuals. It combines a genetic algorithm (GA) with a scan-matching process to solve the SLAM problem. The algorithm is parameterized in terms of accuracy and computational burden. The accuracy of the maps produced varies when different parameter sets are used.

In practice, the SMG-SLAM successfully performs SLAM only when improved parameter sets are used, which significantly increase the algorithm’s computational needs. By exploiting the fine-grain programmable resources of an FPGA and the parallelization potential of the algorithm, a significant acceleration over software can be achieved making the use of improved parameter sets more efficient. This paper focuses on the presentation of a hardware/software system that is implemented in reconfigurable hardware and can provide this acceleration.

The system can be integrated into any mobile robotic platform equipped with range sensors (like Pandora) to provide fast and accurate SLAM without burdening the main processor of the robot. Moreover, because of the low power consumption of reconfigurable hardware compared to general purpose architectures, an FPGA implementation of the SMG-SLAM can prove valuable for a mobile robot that uses batteries.

Novel contributions of this paper include:

- The presentation of the SMG-SLAM algorithm, which is a new solution to the SLAM problem.
- The first, to the authors’ knowledge, hardware implementation of a scan-matching SLAM algorithm that is based on genetic algorithms.

© 2011 Elsevier B.V. All rights reserved.
The proposed SMG-SLAM algorithm uses a genetic process to iteratively match scans taken from an Laser Range Finder (LRF) to the current shape of the global map. The matching process is based on a well-known method called Iterative Closest Point (ICP [26]), which tries to minimize the difference between two clouds of points (usually two consecutive laser scans) by applying a transformation (translation and rotation) to one of the clouds. ICP iteratively revises the transformation in order to minimize the mean square distance between the points of the scans [10,11,26]. In SMG-SLAM, the matching is performed not between two scans (scan-to-scan matching) but between a scan and the current global map (scan-to-map matching). SMG-SLAM matches every new laser scan produced by the LRF to the map it has constructed so far in order to find the robot's translation and rotation since the previous scan. Then, having decided on the new robot position, it updates the map.

2.2. Description

An iteration of the algorithm happens every time a new scan is available by the LRF and comprises three steps:

1. Scan fetching and ray sampling (Section 2.2.1).
2. New pose estimation using a genetic algorithm (Section 2.2.2).
3. Map update (Section 2.2.3).

Step 1 fetches the latest scan data from the sensor and performs some pre-processing. Step 2 uses the data and the current map to decide how much the robot moved since the previous scan. It does this using a genetic algorithm. The result is the new robot pose (from which the latest scan has been taken). Step 3 uses the new pose and the scan data to update the map. Fig. 1 shows the simplified pseudo code of the algorithm. It is divided into several functions (their names in bold) that perform the above three steps.

SMG-SLAM uses an occupancy grid to store the map of the environment, where the value of each cell corresponds to the probability of the cell being free. The cell values range from 0 (the cell is definitely occupied) to 255 (the cell is definitely free). The value 127 represents complete uncertainty over the occupancy of the cell. These values also correspond to gray-scale colors (black for an occupied cell, white for a free one and gray variations for intermediate values). The size of the map is NxN. The robot pose comprises the coordinates of the robot in the two-dimensional (NxN) space \((x,y)\) and the angle between the direction of the robot and the Ox axis \(\theta\).

A detailed description of each step of the algorithm is presented below.

2.2.1. Scan fetching and ray sampling

At the first step of the SMG-SLAM an LRF scan (the only input to the algorithm) is fetched from the sensor. The LRF used was the Sick LMS-200 [8]. The LRF scan consists of 181 pairs of floating point values. Every pair comprises the distance between the robot and a ray's end (i.e. the distances between the robot and the closest obstacle detected by the ray) and the angle of the ray. The scan has an angle range of 180\(^\circ\) so a resolution of one ray per degree is used. Using these distances detected by the laser sensor, the algorithm is able to determine what obstacles are surrounding the robot and how far they are located. This information allows SMG-SLAM to construct a map and position the robot inside this map.

The function \(\text{Fetch \_scan}\) converts the distances from meters (measured by the sensor) to map cells (used in the map) by dividing with a coefficient \((c_{ogd})\). This way, the algorithm is provided with the grid position of the end of each ray in relation to the grid position of the robot.

Because of the fact that nearby rays end up in the same cells in the occupancy grid map, only a subset of the rays is used by the genetic algorithm step to save computing time. A simple “sampling” is performed inside function \(\text{Sample \_rays}\) by choosing one in every \(M\) rays, resulting in \(\frac{181}{M}\) rays. In contrast, the map update step utilizes the full scan vector (181 rays), as will be explained later.

2.2.2. New pose estimation using a genetic algorithm

During the new pose estimation step a genetic algorithm is used. The genes that constitute the genetic population represent possible moves of the robot from its previous pose. They can be seen as transformations (translation \(\Delta x, \Delta y\) in map cells and rotation \(\Delta \theta\) in rad) that could be applied to the previous robot pose to produce the new pose. Each gene is also characterized by a fitness value which shows how likely it is for the gene's transformation to be the correct one (close to reality), taking into account the latest scan. The genetic algorithm’s aim is to evolve the population of genes and come up with a very fit gene, whose transformation matches well with the map and the latest laser scan. The genetic algorithm works like this: At the beginning of the genetic algorithm step, the gene population is initialized with random values (function \(\text{Initialise \_population}\)). These have user-defined ranges (more in Section 2.4). After that, the population evolves from generation to generation trying to preserve the fittest genes and find even better ones. This is done by keeping the best genes and randomly mutating the rest with the hope that better transformations will emerge (function \(\text{Perform \_GA}\)). In every generation of the algorithm, the fitness value of each gene is calculated. Then, the new generation is formed using the elitism and mutation steps. The steps are described in more detail in Sections 2.2.2.1–2.2.2.3.

2.2.2.1. Fitness function

The fitness function (function \(\text{Update \_fitness \_values}\)) finds the fitness value of every gene and for each generation. It applies the gene's suggested transformation to the previous pose, giving the suggested new pose \((x_{new}, y_{new}, \theta_{new})\). Then, it checks how well the laser scan (assuming it had been taken from the suggested new pose) matches the global map. The fitness value of gene \(i\) is the sum of all the map cell values at the sampled rays' ends subtracted from 255:

\[
\text{fitnessMemory}_i = \sum_{j=0}^{181} \left(255 - \text{mapMemory}[i][j] \times Y_{end}(i)[j] \right) \quad (2.1)
\]

In Eq.(2.1) \(j\) is the ray index, \(\text{mapMemory}\) is the map memory, \(\text{fitnessMemory}\) is the fitness memory and \(X_{end}(i)[j], Y_{end}(i)[j]\) are the coordinates of the cell at the end of ray \(j\) for the transformation.
The fitness function is maximized if the scan fits perfectly with the obstacles of the map, namely if the rays’ end cells have a value of 0 (represents an obstacle in the map). The fitness function’s scope is rather straightforward: If the scan matches the

```python
function SlamIteration
    Fetch_scan
        for (i=0; i<180, step=1)
            Fetch (d(i),a(i)) pair and store it in memory
            Compute coordinates of ray i end, convert from meters to map cells
        endfor
    Sample_rays
        for (i=0; i<180; step=M)
            Save ray i to Scan Memory
        endfor
    Initialize_population
        for (i=0; i<P; step=1)
            Randomly initialize gene i (Δx,Δy,Δθ)
        endfor
    Perform_GA
        for (generationIndex=0; generationIndex<G; step=1)
            Update_fitness_values
                for (i=0; i<P; step=1)
                    Fitness of ith gene = 0
                    Apply ith gene’s transformation (Δx,Δy,Δθ) to robot pose to get proposed pose (x_new,y_new,θ_new)
                    Compute sin(θ_new) and cos(θ_new)
                    for (j=0; j=M; step=1)
                        Compute jth ray’s end coordinates (x_end,y_end)
                        Use 2.1 to update fitness of gene i
                    endfor
                    if (Best fitness<Fitness of gene i)
                        Best fitness = Fitness of gene i
                        Best gene = gene i
                    endif
                endfor
            Elitism
                Copy Best gene to next generation
                for (i=1; i<P; step=1)
                    Create a mutated copy of Best gene, make it the ith gene of the next generation
                endfor
            Mutation
                for (i=P; i<P; step=1)
                    Pick a gene of the current generation using tournament selection
                    Create a mutated copy of the gene, make it the ith gene of the next generation
                endfor
        endfor
    Update_pose
        Apply transformation of fittest gene to get the new pose
    Update_map
        for (rayId=0; rayId<180; step=1)
            Get length d of ray rayId
            for (R=1; R<maximumRayRange; step=1)
                Compute coordinates of Rth cell across ray rayId
                if (d>R)
                Read Rth cell’s value from map memory and update using (2.2)
                else if (d>R-1 AND (d<2R))
                Read Rth cell’s value from map memory and update using (2.3)
                endif
            endfor
        endfor
end SlamIteration
```

**Fig. 1.** Pseudo code of SMG-SLAM (simplified).
obstacles in the map after the gene's transformation has been applied, the gene gets a high fitness value. If the scan falls in unexplored or unoccupied space after the gene's transformation has been applied, the gene gets a low fitness value. The function also keeps track of the fittest gene and its fitness at every time step.

When all the fitness values of the genes have been calculated for the current generation, they can be fed to the elitism and mutation functions to form the new generation.

2.2.2.2. Elitism. In general, elitism is used in order to make certain that the best gene so far is kept in the next generation's population. In SMG-SLAM, the function Elitism performs a two-stage elitism process:

- The gene with the best fitness value passes unmodified to the next generation.
- A few mutated copies of the best gene are also transmitted to the next generation. Mutation is done by adding or subtracting small random values to the best gene's translation and rotation. These elite genes constitute $e\%$ of the total genes of the new population, a parameter selected by the user.

2.2.2.3. Mutation. The rest of the new generation ($mu = (1 - e)\%$) is produced by mutating genes of the current generation (function Mutation). Mutation is performed using tournament selection (pick two genes and compare their fitness values). The winner of the tournament is mutated in the same way described in Section 2.2.2.2 and then passed to the next generation.

After the genetic algorithm terminates (i.e. the predefined number of generations has been processed), the best gene's transformation is applied to the previous pose to get the new pose estimate (function Update_pose).

2.2.3. Map update

After the pose estimation step the new pose of the robot is known and the map update step can begin (function Update_map). The algorithm now knows the position from which the current scan has been taken. Thus it can use the scan to improve the map. Only the cells that are in the conceptual field of the LRF (half a disc) are updated. For each of the 181 scan rays, the cells placed along the ray are examined and updated. The function computes the coordinates of each map cell, reads its value and updates it. The update is done by first comparing the cell's distance from the robot according to the current map with the cell's distance from the robot according to the scan. Depending on the result of the comparison one of three actions is taken (keeping in mind that a cell's value corresponds to the probability of the cell to be free):

- If the cell is between the robot and the obstacle (the cell's distance from the robot according to the map is smaller than the detected distance of the laser ray), its value (the probability of it being unoccupied) is increased by a factor proportional to the difference of the cell's previous value from the value 255. More specifically, the difference is multiplied by a coefficient inc:

$$new = old + (255 - old) \times inc \tag{2.2}$$

- If the cell is behind the obstacle (the distance of the cell from the robot is greater than the corresponding ray's distance), the cell is out of the conceptual field of the LRF so its value remains unchanged.

2.3. Accuracy issues and proposed enhancements

The accumulation of rotational error during pose estimation is responsible for multiple appearances of the same obstacles in different positions during loop closure. A technique that was used in order to minimize the cumulative rotational error was the runtime modification of the map update coefficients $inc$, $dec$ depending on the motion of the robot. Generally, small values of the map update coefficients are preferred in order to minimize the effect of each scan on the map and allow the algorithm to gradually become confident about the form of the map and correct small errors in estimation. This strategy works well when the robot moves in a straight or almost straight line, when the robot observes the same obstacles for a long time and is able to eventually find the best solution.

Nevertheless, when the robot has a large rotational speed in comparison to its linear speed, it does not have a lot of time to become confident about the shape of the map. The slow update strategy causes repetitive drawing of the same corridor in slightly different situations (accumulation of error). In these situations, increased values of the update coefficients are necessary. They make the algorithm update map values more rapidly. This means that the algorithm quickly forms a confident estimation of the environment, not allowing subsequent updates deform the map easily and accumulate rotational error. In practice, this strategy is implemented by checking the last rotation of the robot before updating a map value (see inside Update_map in Fig. 1). If the last rotation is above a certain threshold (rot), then increased values of the update coefficient are used. Otherwise, more conservative values are chosen.

Another issue with SMG-SLAM is its inefficiency in finding the correct solution in a feature free environment, meaning an environment without corners or significant changes in its topology. For example, when the robot is moving in a homogenous corridor, the algorithm cannot calculate the transformation of the robot's pose correctly because the scan can be matched to many different places in the corridor. In order to partially fix this problem, previous moves of the robot (called "momentum") were used. The latest moves (transformations) of the robot are stored in memory and a few genes corresponding to these transformations are added to the genetic population ($mu$ of the new population, not shown in Fig. 1). It was assumed that the transformations of the robot when moving in a corridor would be similar to its previous transformations due to the robot's momentum, since it is not possible for the robot to instantly change its movement from linear to rotational.

2.4. Choice of parameters and its impact on performance and accuracy

There are several adjustable parameters in SMG-SLAM. Some of them have to be chosen carefully to optimize the accuracy of the results and others can be used to trade accuracy for speed and vice versa. The main parameters and their impact on the algorithm are described below:

- $N$: Defines the size of the map in terms of the number of cells. A larger map allows for exploration of larger areas but requires more memory. This might be a problem in a hardware implementation, as will be described in Section 4. Larger $N$'s are also needed when the resolution of the map increases (see eqd). For the problems tested in this paper, $N = 512$ cells or $N = 724$ cells was used.
ocgd: The resolution of the map is defined by the ocgd parameter, measured in meters/map cells. When ocgd is decreased, the area represented by each cell is decreased as well and this results in finer map resolution. This improves the SLAM accuracy with the cost of larger memory requirements. When ocgd is increased, the map resolution becomes coarser, saving memory resources and limiting SLAM accuracy. The effect of ocgd on accuracy is illustrated in Section 2. The performance of the SMG-SLAM is also affected by ocgd, since smaller ocgd values mean more cells have to be updated in every map update. The usual values of ocgd range from 0.02 to 0.12 m/map cell. These values were chosen taking into account the range of the laser sensor (8 m) and the memory and computational needs.

$M$: This parameter defines the density of the ray sampling process (1 in every $M$ rays is sampled). Only some rays are used during fitness evaluation. The choice affects the accuracy of the algorithm, since more rays (smaller $M$) mean a safer fitness choice for every gene. The benefits decline when $M$ becomes very small, since neighboring rays start falling on the same map cells. $M$ is also crucial for performance, since fitness evaluation is costly. By doubling $M$, the execution time is reduced almost by 45%. Values between 3 and 12 rays$^{-1}$ were used, depending on the desired accuracy/performance combination.

$P$: The size of the genetic population. This parameter influences accuracy but not as much as $M$ does. It is necessary to use a large enough population (usually more than 700 genes) to achieve satisfactory results. The execution time is almost directly proportional to $P$, which motivates the use of smaller populations.

$G$: The number of genetic generations during one iteration of the algorithm. This parameter also influences execution time. Generations over a certain value (40) provide no benefit. Practically, 20–40 generations were used in the experiments.

**Initialization range** $(\Delta x_{\text{init}}, \Delta y_{\text{init}}, \Delta h_{\text{init}})$: These are the random values given to each gene during initialization. Their ranges are:

- $-4 \leq \Delta x_{\text{init}} \leq 4$
- $-4 \leq \Delta y_{\text{init}} \leq 4$
- $-0.125 \leq \Delta h_{\text{init}} \leq 0.125$

These values come from empirical observations, which showed that the majority of successful gene values are inside these ranges.

**Mutation range**: $(d x_{\text{random}}, d y_{\text{random}}, d h_{\text{random}})$ in mutation and elitism: These are random values added to the gene during mutation or elitism. Their ranges are narrower than those of $(\Delta x_{\text{init}}, \Delta y_{\text{init}}, \Delta h_{\text{init}})$.

**Elitism, momentum and mutation percentages** ($e, m o, m u$): In each generation, $e$% of the genes come from the elitism operation, $m o$% come from the momentum operation and $m u$% come from the mutation operation. Elitism is usually chosen for the first few genes ($1\% \leq e \leq 2\%$). A few genes are enough to transfer information about the elite genes to the next generation. The same applies to the momentum operation. Thus, mutation is the dominant operation ($96\% \leq m u < 98\%$). It is the main source of new genes that could have high fitness values.

**Map update parameters**: (inc, dec, rot): The effect of the above parameters is described in Section 2.3. The values used for inc and dec are between 0.01 and 0.5, depending on the problem. The value of rot is usually set to 0.2 rad. Tuning might be necessary to find the best configuration. In Fig. 1, the different values of the coefficients are represented as inc1, inc2, dec1, dec2.

### 2.5. Testing

To test the behavior of the robot and the LRF in various environments, the Unified System for Automation and Robot Simulation (USARSim) simulator in collaboration with the Player robot device interface were used. USARSim is a high-fidelity simulator of robots and environments based on the Unreal Tournament game engine [23]. Player is a network server for robot control that supports a variety of robot hardware [15].

### 3. Related work

Reconfigurable hardware implementations of SLAM algorithms are limited and none of them is based on genetic algorithms. Most efforts are based on visual sensors (i.e. cameras).

In [3], an FPGA implementation of the EKF algorithm is presented and used to perform SLAM. This approach is different than the one presented here. Multiple cameras are used to extract features (i.e. landmarks) from the environment and then to feature locations and the robot’s pose are updated using the EKF. In contrast, the SMG-SLAM algorithm uses laser measurements and is not feature-based. It just matches the scan with the map point by point. The authors report a threefold acceleration compared to a desktop processor and a 13-fold acceleration compared to an ARM embedded processor. The acceleration is based on the parallelization of matrix operations (which are very demanding in EKFs with thousands of features). In the hardware implementation of the SMG-SLAM, the acceleration comes from parallelizing the fitness update step of the genetic algorithm.

In [16], a system that performs feature-based SLAM using an odometric sensor and a stereo camera is developed. The system is partitioned into hardware and software. An FPGA extracts the features from the sensor data and a processor executes the rest of the SLAM operations. This work mainly addresses the inability of software to extract the features in real time. The implementation of the SMG-SLAM that is proposed here also uses hardware/software co-design but mainly focuses on the hardware part of the implementation. Also, it explores the acceleration of the complete SMG-SLAM algorithm and does not need to extract features.

Some work on accelerating parts of scan-matching SLAM algorithms using FPGAs have also appeared in the literature. A hardware feature detector for stereo images is proposed in [7]. This detector can be placed between the 3D sensors and the SLAM algorithm. A similar approach is presented in [25]. Neither of these works focuses on the SLAM algorithm as a whole but rather on how the computationally intensive feature extraction from stereo images can be accelerated using dedicated hardware.

Because the genetic algorithm is the core of the SMG-SLAM, a summary of work related to hardware implementations of genetic algorithms will also be presented here. Genetic algorithms are ideal for hardware parallelization due to the need for the same computation to be performed for all the genes of the population. The subject has attracted the interest of researchers because of the increased computational needs of genetic algorithms and their widespread use. Most authors focus on the implementation of a generic architecture without considering the needs of a specific application (e.g. map memory access in SMG-SLAM).

In [13], the authors suggest the use of a neural network to evaluate the genes’ fitness values. Their implementation is massively parallel (selection, crossover and mutation are performed in parallel for all of the genes). Their solution is application independent, since the neural network can be used to compute any kind of fitness function. Nevertheless, the neural network needs to be trained before it can be used and this assumes that the fitness evaluation only depends on the gene’s value. In SMG-SLAM this is not the case, since the fitness evaluation depends on map values that correspond to the gene. Also, the massive parallelism proposed is impossible if the fitness evaluation requires reading values from memory, like in SMG-SLAM.

In [4], a solution with a smaller amount of parallelization is proposed that can choose between a number of different fitness
functions and is also fully parameterized. The architecture of the system is based on a number of modules that operate in a serial manner and two RAMs used to store the genes selected for crossover/mutation and the future population. The system is applied to the Travelling Salesman Problem (TSP) for testing purposes. In [17], the authors use the Island Model GA (IGA) technique to achieve parallelization. The genetic population is divided into several islands, in each of which genetic evolution happens independently. Every island has a pipelined architecture that consists of selection, crossover and mutation modules. In each clock cycle, one gene is updated. A small amount of genes is periodically copied from each island to the neighboring islands so as to avoid convergence to local maxima.

In [21], the authors suggest the use of a 197-gene population that is always sorted by fitness value. In each clock cycle, three new genes are produced and are inserted in the population in the right places. After that, the three less fit genes are deleted and the process is repeated. A large number of generations are needed for convergence to an optimal solution and parallelization is achieved by using multiple identical copies of the population that allow for simultaneous access to several genes. This approach might be memory intensive for applications with large population sizes (like the ones needed in SMG-SLAM).

In [20], a software/hardware system is implemented using an FPGA and a desktop computer. A processor inside the FPGA is used for communication with the desktop computer and control of the hardware system. A selection of six different fitness functions is provided and other parameters of the system are also adaptable. Pipelined multipliers are used to implement the fitness functions and two RAMs (one for the previous and one for the next generation) are employed.

In [6], a generic hardware core (IP – Intellectual Property) that can be used as a genetic module in a bigger system is proposed. The design provides full programmability and easy interfacing. The genetic algorithm’s parameters can be set externally, using handshaking signals. No specific fitness module is implemented. Instead, an interface is provided so that external fitness modules can be implemented and connected to the IP. Up to eight different fitness modules can be multiplexed.

4. Proposed design

4.1. Software profiling and hardware/software partitioning

Profiling the software algorithm is necessary before transferring it to hardware. Through profiling, information is acquired over which functions of the algorithm are the most computationally intensive as well as which computations inside these functions are performed most often. This information can be used to decide which parts of the system should be implemented in hardware (logic blocks of the FPGA) and which ones can be kept in software (run on the FPGA’s embedded processor) without significant loss in overall performance. Table 1 shows the profiling results of a typical execution of the algorithm, taking as input a sequence of 207 laser scans and using the parameter values shown (only the most important functions are shown). More demanding parameter sets (e.g., \( P = 2000 \), \( M = 10 \)) give similar results.

It is clear that the Update_fitness_values function is the most demanding by far and takes up more of the processor’s time as the accuracy of the algorithm increases. The Elitism, Mutation, Momentum and Update_map functions are less demanding.

In line with the above results, it was decided that the whole genetic part of the algorithm (Initialize_population, Update_fitness_values, Elitism, Mutation, Momentum functions) should be implemented in hardware. The Update_fitness_values function is the main bottleneck that has to be accelerated to speed up the SLAM process. The other functions of the genetic part were also implemented in hardware, since they must all access the same population and fitness memories that the Update_fitness_values function uses (these memories were implemented with Block RAMs of the FPGA as will be described later). Keeping these functions in software would require transferring data from software to hardware and vice versa in each iteration and would place a significant overhead in the process.

As for the Update_map function and the map memory, there are two main approaches that can be considered. First, the map can be stored in an external memory and the Update_map function can be executed in hardware. This approach allows for large and adaptable map sizes to be used. Nevertheless, it has an important drawback: Several map cell values have to be transferred from the external memory to the fitness update hardware module for each gene whose fitness is evaluated, since the evaluation is done using Eq. (2.1). This places an overhead to the system that can easily overshadow the acceleration of other modules implemented in hardware. Moreover, the map cell coordinate values span over a large map area (due to the 180° wide laser scan) and are random due to the unknown environment. Thus, making a bulk data transfer of all the surrounding cells that might be needed before each scan would also be inefficient; too many cell values would have to be transferred.

In order to avoid the numerous data transfers, a different approach was used. The map memory was implemented with BlockRAMs. Thus, the hardware modules can access the map memory in only one clock cycle. Nevertheless, the implementation of the map memory with BlockRAMs places an upper limit in the size of the map that can be used, since FPGA memory resources are limited. Generally, this means that the resolution of the map (inversely proportional to the \( c_{gd} \) parameter) has to be decreased in order to store large maps in the memory. This reduces the accuracy of the algorithm as will be shown later (Section 6). For practical implementations, a modern FPGA with numerous BlockRAMs (e.g. a Xilinx Virtex-5) has to be used.

The remaining parts of the algorithm (Fetch_scan function and some basic input/output functions) were kept in software to simplify access to the LRF and interconnection with the rest of the robotic system.

4.2. Choice of word-lengths

When the algorithm runs on a CPU, integer and double-precision floating point data types are used for the variables and coefficients. When designing hardware this is not the case. FPGA implementations benefit a lot from the use of fixed-point instead of floating-point arithmetic, since floating-point arithmetic operators are expensive to implement. The range of variables also plays a role in selecting the most suitable arithmetic representation. Fixed-point numbers that span large ranges require too many bits and in these cases floating-point is preferable. If ranges are limited, fixed-point is the best solution.

As will become apparent, the ranges of the SMG-SLAM variables are extremely small, allowing for the use of fixed-point arithmetic.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Percentage of processor time used (%)</th>
<th>Number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch_scan</td>
<td>0.10</td>
<td>207</td>
</tr>
<tr>
<td>Initialize_population</td>
<td>0.10</td>
<td>207</td>
</tr>
<tr>
<td>Update_fitness_values</td>
<td>92.62</td>
<td>6210</td>
</tr>
<tr>
<td>Elitism, mutation, momentum</td>
<td>2.97</td>
<td>6210</td>
</tr>
<tr>
<td>Update_map</td>
<td>4.21</td>
<td>207</td>
</tr>
</tbody>
</table>
Even with fixed-point, it is desirable to reduce the word-lengths of the variables as much as possible, in order to make operators faster and more resource-efficient. Of course, fewer bits mean lower precision, thus problems might be caused to the algorithm’s accuracy. In what follows, the choice of word-length for each of the main variables of SMG-SLAM will be discussed. These choices are guided by the types of problems and parameter sets used in the experiments (Section 2.4). They can be altered easily to support different configurations.

Robot pose (x,y): Unsigned integers in software, ranging from 0 to N (maximum 723 for the maps used). A 10-bit fixed-point representation was used in hardware.

Robot pose (\theta): Double precision floating-point software variable taking all possible real values, since the rotation angle is accumulated. For example, if the robot rotates clock-wise for an infinite period, \theta will grow towards \( +\infty \). A 16-bit fixed-point representation was used in the hardware system. 6 bits (including 1 sign bit) were used for the integer part and 10 bits for the fractional part. The 6 integer bits are provided so that any practically possible angle can be represented. In fact, the robot’s absolute angle never reached a value higher than 17. The 10 fractional bits allow for a resolution of 9.765 \times 10^{-4} \text{ rad or 0.055^\circ} . This was the smallest resolution for which rotational error was not accumulated rapidly. For fewer fractional bits, the robot’s rotation during the hardware run could not “follow” the robot’s rotation during the software run (all other parameters and maps being the same).

Gene values (\Delta x, \Delta y): Integers in software that can take any positive or negative value (they count map cells). Practically, they are initialized in the range \([-4,4]\) and most genes have small \Delta x and \Delta y values in all test runs that were made. \Delta x and \Delta y never exceeded the \([-9,9]\) range. Thus, in hardware, a 5-bit fixed-point variable was used to represent each one of \Delta x and \Delta y (1 sign bit and 4 integer bits).

Gene values (\Delta \theta): Double precision floating-point value in software that can take any real value (it is expressed in rad). Practically, \Delta \theta is initialized in the range \([-0.125,0.125]\) and almost all genes have small \Delta \theta values (absolute value less than 1). Nevertheless, sudden rotations happen occasionally. This is why a 14-bit fixed-point variable was used in the hardware system. 4 bits (including 1 sign bit) were used for the integer part and 10 bits for the fractional part (for the same reasons described for \theta above).

Fitness values Unsigned integers in software, ranging from 0 (if all rays end at definitely unoccupied cells) to \( \frac{M}{\Delta} \times 225 \) (if all rays end at definitely occupied cells). For the usual configurations, where \( M \geq 3 \), the fitness value can take values from 0 to 15,300. A 14-bit fixed-point hardware representation was used to cover all possible values.

Ray lengths (d): Unsigned integer (representing map cells) in software, ranging from 0 (ray’s length 0, obstacle “touches” sensor) to the maximum distance visible by the sensor. The LRF has an 8-m range. This corresponds to a maximum of 400 cells for the \text{ocgd} values used. A 10-bit fixed-point representation was used in hardware to cover a possible future expansion.

Ray end coordinates (x_{scan}, y_{scan}): Integers in software, ranging from \(-N\) to \( N \) (representing map cells). A 10-bit fixed-point representation was used in hardware.

Trigonometric variables (\sin \theta, \cos \theta): Double precision floating-point variables in software, in the range \([-1,1]\). A 12-bit fixed-point representation was used (2 bits integer, 10 bits fractional part) in hardware.

4.3. System architecture

The FPGA-based architecture of the system is shown in Fig. 2. It comprises an embedded PowerPC 405 processor that executes the software functions and a custom peripheral that implements the hardware functions. Both parts are placed inside the FPGA. A fast PLB bus (Processor Local Bus [1]) is used for communication between the processor and the custom peripheral and a slow OPB bus (On-chip Peripheral Bus [14]) is used to connect the LRF and the processor.

4.3.1. Software

Every time a new scan (a vector of 181 floating point distance values, \text{laserDistance}) becomes available, the processor fetches it through the OPB bus and pre-processes it so that it can be accessed by the hardware part (essentially running the \text{Fetch_scan} function). For every \( k \), the processor produces a triplet of values \( d[k], x_{scan} - k, y_{scan}[k] \) (converted to map cells by dividing with \text{ocgd} ) and every value is rounded off to the closest integer, since the hardware part represents these values as 10-bit integers (Section 4.2). Then the 181 triplets are transferred to a 181 \times 30\) bits scan memory inside the custom peripheral through the PLB bus (using a DMA module). They are used to execute the next iteration of the algorithm (update the pose and the map).
4.3.2. Hardware

All hardware modules are placed inside the custom peripheral. Fig. 3 shows the various modules that comprise the custom peripheral and the interconnections between them.

The DMA module is responsible for the transfer of scan triplets from the processor to the Scan Array memory (using the Slave Interface) after every new scan is fetched by the processor. Two software addressable registers (not shown in Fig. 3 for simplicity) are used for processor-hardware handshaking. The controller module coordinates the operation of the other modules, activating and deactivating them when necessary and giving the active module access to the memories that are related to its operation. The setup module initializes the map after a system reset. The ray sampling module chooses the triplets that will be used for fitness evaluation. The initialization module initializes the genetic population with random gene values at the start of the iteration. The fitness update module evaluates each gene’s fitness using scan triplets and map values. The elitism–mutation–momentum (EMM) module performs elitism mutation and the momentum operation to the current gene population and produces a new population. The map update module updates the map at the end of each iteration.

There are five separate memories inside the peripheral which can be accessed by all the above modules. The population memory stores the P genes of the population (every gene’s value comprises a relative translation \( \Delta x, \Delta y \) and a relative rotation \( \Delta \theta \)). The fitness memory stores the fitness values of each gene. The map memory’s elements correspond to map cell values. The best gene register (not shown in Fig. 3) stores the fittest gene of the population at each moment. Finally, the scan array memory is used to store the 181 triplets sent by the processor, as described earlier.

For every scan that is fetched from the processor, a new iteration of the algorithm begins. The modules are activated as follows:

1. The initialization module initializes the genetic population
2. The ray sampling module chooses the triplets that will be used for fitness evaluation
3. Genes’ fitness values are evaluated by the fitness update module
4. The EMM module creates a new population for the next generation
5. If the generation counter has not reached a user-defined maximum value \( G \), the algorithm returns to step 3 to begin a new generation
6. The map update module updates the map cells
7. The iteration has finished. A new pose and an updated version of the map are available. The system now waits for a new scan to be fetched from the processor.

The architecture of each module is described in Sections 4.3.2.1–4.3.2.6.
4.3.2.3. Setup, initialization and ray sampling modules. The setup module stores and updates the robot pose and the generations counter.

4.3.2.4. Fitness update module. The fitness update is activated once for every generation and computes the fitness values of all the genes of the population. The complete fitness update function can be found in Fig. 1. There is an outer loop that examines one gene per iteration and an inner loop that examines one laser ray per iteration. The most critical part is the inner loop, where four multiplications are performed to calculate the map coordinates at the end of each ray.

4.3.2.1. Controller module. The Controller module is based on a 12-state FSM (Finite State Machine). The tasks performed during each state can be found in Table 2. The Controller module uses handshaking signals to activate the modules and connects them to the appropriate RAMs (only the active module has access to RAMs). Finally, the Controller module stores and updates the robot pose and the generations counter.

4.3.2.2. Population, fitness, map and scan array modules. The population and fitness memories are both dual-port and have equal size \( P \). They are implemented using Block RAMs of the FPGA. The population memory has a width of 24 bits, representing a gene's suggested translation and rotation and the Fitness memory has a width of 14 bits. The Map memory is a dual-port Block RAM of size \( N^2 \) and width of 8 bits. The Scan Array memory is a 181 × 30 single-port Block RAM that has its own address field and can be accessed directly by the processor. Each triplet stored in the array comprises the values the ray's length and the coordinates of the ray's end. A DMA module is used in burst transfer mode to write all the triplets to the array every time a new scan is produced.

4.3.2.3. Setup, initialization and ray sampling modules. The setup module is activated after a system reset. It initializes the map, writing the value 127 to all the map cells. The process lasts \( N^2 \) clock cycles.

The Initialization module generates random gene values that are written to the population memory. This way, the genetic population includes a variety of possible relative movements. During the run of the genetic algorithm, the population gradually improves its overall fitness by keeping only the fittest genes that correspond to the most likely relative movement and mutating the rest.

To generate the random gene values, a 24-bit LFSR (Linear Feedback Shift Register) is used. LFSRs are a very common type of pseudorandom number generator used in hardware implementations. The 24-bit LFSR was implemented using a programmable Xilinx IP core [9]. The particular implementation is used because it consumes less FPGA resources than other solutions, it can be easily customized in terms of bit-width and other parameters and its seed value can be updated at runtime. The 24-bit output of the LFSR is manipulated such that each one of its three components \((Ax_{init}, Ay_{init}, A0_{init})\) uniformly fall within the desired initialization range. The 5-bit \(Ax_{init}\) and \(Ay_{init}\) parts are right-shifted twice and then sign-extended to remain in the range \([-4, 4]\) and the \(A0_{init}\) part is right-shifted seven times and then sign-extended to remain in the range \([-0.125, 0.125]\). Then the values are forwarded to the population memory.

The module is pipelined in order to generate one gene per clock cycle (e.g. the LFSR has 16 pipeline stages). \( P \) cycles are needed for the whole population to be initialized.

The Ray Sampling module reads the Scan Array memory and samples one in every \( M \) triplets to be used during the fitness update. Then it sends the triplets to the fitness update module. The whole process takes \( 180/M \) clock cycles (one clock cycle per triplet) and the logic required is very simple.

<p>| States of the FSM and tasks performed in each one. |</p>
<table>
<thead>
<tr>
<th>State</th>
<th>Task performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>The map memory is initialized</td>
</tr>
<tr>
<td>stby</td>
<td>The controller waits for a new scan</td>
</tr>
<tr>
<td>init</td>
<td>The genetic population is initialized with random values</td>
</tr>
<tr>
<td>sample</td>
<td>The necessary triplets for fitness evaluation are copied from the Scan Array memory</td>
</tr>
<tr>
<td>cntr_0</td>
<td>The generation counter is set to zero</td>
</tr>
<tr>
<td>best_0</td>
<td>The best gene register is erased</td>
</tr>
<tr>
<td>fit</td>
<td>Fitness values of the entire population are evaluated</td>
</tr>
<tr>
<td>emm</td>
<td>The new generation's population is produced performing elitism, mutation and adding momentum genes</td>
</tr>
<tr>
<td>cntr_Inc</td>
<td>The generations counter is increased by 1</td>
</tr>
<tr>
<td>cntr_check</td>
<td>The controller checks whether the generations counter has reached its maximum value ( G )</td>
</tr>
<tr>
<td>pose</td>
<td>The robot pose is updated using the best gene</td>
</tr>
<tr>
<td>map</td>
<td>The map is updated</td>
</tr>
</tbody>
</table>

The module is connected to the population, fitness and map memories. It comprises two blocks, as seen in Fig. 4, which is a simplified block diagram of the module's architecture. The first block (BLOCK 1) is responsible for the fetching of the genes from the Population memory and the calculation of the map addresses that need to be read. This is the most computationally demanding task, since it requires many multiplications and has to be performed many times for every gene. The second block (BLOCK 2) uses the map values that correspond to the addresses generated by the upper block to calculate each gene's fitness value.

The first block first reads a gene and adds its \( \Delta v \) value to the current \( v \) of the robot. The result is passed to a CORDIC (Coordinate Rotation Digital Computer [22]) module which outputs the corresponding \( \sin v \) and \( \cos v \) values (12 bits each). No new gene value is read (and thus \( \sin v, \cos v \) remain constant) until all the rays of the gene have been examined. Also, the block reads the pairs \((x_{scan}, y_{scan})\) and \((x_{new}, y_{new})\) (10 bits each) from the scan memory, which correspond to two rays of the current scan. These pairs are updated with every clock cycle, feeding the critical path of the design. The \( \sin v, \cos v, x_{scan}, y_{scan}, x_{new}, y_{new} \) values are used as inputs to the eight DSP blocks that perform \( 10 \times 12 \) fixed-point multiplication. The outputs are used together with the \( x_{new}, y_{new} \) values of the examined gene to generate two pairs of coordinates. These are the coordinates of the map cells on which the two examined rays have detected an obstacle. They are converted to map memory addresses using a shifter and an adder and sent to the map memory (one pair every clock cycle).

All the operators that constitute the critical path of the design are pipelined in order to maximize the throughput.

The second block receives a pair of cell values in every clock cycle and implements Eq. (2.1). It uses an accumulator to calculate the fitness of the examined gene. The block also keeps track of the best gene (not depicted in Fig. 4).

The parallelism comes from the fact that the eight multiplications are performed simultaneously (and thus two rays are examined in parallel). In theory, all \( 180 \) rays could be examined in parallel (all inner loop calculations in Fig. 1) if the target FPGA has enough resources. In fact, even multiple genes could be processed in parallel since the fitness evaluations are independent. In practice, though, the dual-port map memory is a bottleneck. One map read is necessary to examine every ray and only two reads can be performed in the same clock cycle using a dual-port memory. Still, the speedup that is achieved is significant. This is because:

a) Pipelining is used: Essentially, two parallel pipelines are employed, one for every ray. By filling the pipelines, it is guaranteed that two map addresses are generated in every clock cycle and thus two map cell values are read from the map memory. Every gene fitness evaluation needs \( 12 \) map
cell values which is equivalent to \( \frac{180}{M} \) clock cycles. For typical values of \( M \), this means that 3–4 genes are inside the pipelines at any given time.

b) Inside every pipeline, many components work in parallel: Eight multiplications are performed simultaneously, together with several other operations.

The use of a CORDIC module to calculate the trigonometric functions was decided because it uses few slices and no multipliers. Its latency is worse than other approaches but this is irrelevant here, since the module is not on the critical path of the hardware implementation. This is not the case for the multiplications, which are critical for the performance of the design and thus fast DSP blocks were used to implement them.

A total of \( \frac{180}{M} \cdot P \) clock cycles are needed for the module to evaluate all fitnesses.

4.3.2.5. Elitism–mutation–momentum (EMM) module. This module generates a gene population for the new generation after the fitness update module has finished. It produces one new gene in every clock cycle using one of three different operations: Elitism, momentum and mutation.

The module’s architecture is very simple. Two LFSRs are used, one to produce random memory addresses for the mutation operation and one to produce mutation values used in all operations. Which operation will produce the gene is decided by comparing the population counter to predefined constants and activating the appropriate block. During the elitism and momentum steps, the module simply adds the random values to the fittest and momentum genes and passes the result to the population memory. During the mutation step, two genes and their fitness values are read in each clock cycle. The fitness values are compared and the result of the mutation is written to the population memory. \( P \) clock cycles are needed for the module to create the new population.

4.3.2.6. Map update module. The map update module is activated once the genetic algorithm has terminated and the new robot pose is known. The module uses the new pose and the measurement triplets from the Scan Array memory to update all the map cell values inside the robot’s conceptual field, as seen in Fig. 1. The outer loop is repeated for all the rays and the inner loop for every point along a ray. The critical part is the calculation of the coordinates of the cell that has to be updated. The module comprises two blocks. The first block calculates the coordinates of the examined map cell and fetches its value and the second block computes the updated cell value.

In the first block, the measurement triplets for every ray are read and a CORDIC module is used to calculate the trigonometric values inside function \( Update\_map \) of Fig. 1 (these values remain constant for all points along one ray). In every clock cycle, a new pair of coordinates is calculated. An accumulator is used instead of a multiplier, since \( R \) is always increased by 1.

The second block checks the two criteria described in Sections 2.2.3 and 2.3 (position of cell compared to obstacle and rotation angle) and uses a multiplexer to pick the right constant by which the old value will be multiplied. The constants are chosen so that the multiplications can be implementing with shifters (in order

![Architecture of the fitness update module (simplified).](image-url)
to avoid using more DSP blocks). The updated value is written back to the map memory.

The speedup in this module is provided by using a long pipeline that allows one cell to be updated per clock cycle. Again, the dual-port map memory does not allow for more parallelism, since one read operation has be performed in every cycle apart from the write operation. The overall number of cycles needed to update the map is \((8/\text{ocgd}) = 181\).

5. Synthesis results

The hardware code was written in VHDL. It is parameterized in terms of map size \((N)\), map resolution \((\text{ocgd})\), genetic population size \((P)\) and number of rays used for fitness evaluation \((M)\) when \(N = 724\), \(M = 5\), \(\text{ocgd} = 0.07\) and a population of 2000 genes \((P = 2000)\) is used. The targeted FPGA is a Xilinx Virtex-5 XC5VFX70T. A maximum internal frequency of 143.394 MHz is achieved while resource utilization is low except for the number of Block RAMs used. Table 4 shows the distribution of resources among the various modules. The fitness update and map update module consume a large chunk of the total slices and all of the total DSP blocks needed for the implementation of the system, since they contain most of the logic. Most of the BlockRAMs are used to store the map. If a different parameter combination is used, Block RAM utilization will change. With \(N = 512\), \(M = 5\), \(\text{ocgd} = 0.10\) and \(P = 2000\), only 67 Block RAMs will be used (45%).

6. Evaluation

In order to evaluate the performance of the proposed hardware system, the accuracy of solutions to the SLAM problem (estimated pose and map) and the speed achieved were compared to the accuracy and speed of the software algorithm. The hardware system was simulated using Mentor Graphics ModelSim and the software algorithm was written and tested in C++. In both cases, scans were read from a disk file. Several different files were used, each one corresponding to the exploration of a different terrain. The terrains were designed and explored using the USARSim and Player tools mentioned earlier.

The following four parameter sets were used in simulation:

- \(P = 500\), \(M = 9\), \(G = 25\), \(\text{ocgd} = 0.10\) (basic version)
- \(P = 500\), \(M = 9\), \(G = 25\), \(\text{ocgd} = 0.07\) (increased map resolution)
- \(P = 2000\), \(M = 5\), \(G = 25\), \(\text{ocgd} = 0.10\) (decreased \(M\) and increased population size)
- \(P = 2000\), \(M = 5\), \(G = 25\), \(\text{ocgd} = 0.07\) (top version – decreased \(M\), increased map resolution and population size)

6.1. Accuracy

Figs. 5–10 compare the maps produced by software and hardware runs of the algorithm for different parameter sets and terrains. Specifically, three different terrains are used, the corner, the circle and the rooms terrain. The results for the basic and the top version of the system are depicted for each case.

The quality of the SLAM solution is similar in the hardware and software versions in all cases. Differences in the shape of the map

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Peripheral synthesis report.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of resource</strong></td>
<td><strong>Available</strong></td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td><strong>Number of slice registers</strong> 3203 44,800 7%</td>
</tr>
<tr>
<td><strong>Number of block RAMs</strong> 131 148 89%</td>
<td><strong>Number of DSP48E blocks</strong> 9 128 7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4</th>
<th>Distribution of resources among modules.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module</strong></td>
<td><strong>Slice registers used (% overall)</strong></td>
</tr>
<tr>
<td>Controller</td>
<td>161 (5.0%)</td>
</tr>
<tr>
<td>Initialization</td>
<td>100 (3.1%)</td>
</tr>
<tr>
<td>Fitness update</td>
<td>1061 (33.1%)</td>
</tr>
<tr>
<td>EMM</td>
<td>217 (6.8%)</td>
</tr>
<tr>
<td>Map update</td>
<td>1236 (38.6%)</td>
</tr>
<tr>
<td>Population and fitness memories</td>
<td>–</td>
</tr>
<tr>
<td>Map memory</td>
<td>–</td>
</tr>
<tr>
<td>Others</td>
<td>428 (13.4%)</td>
</tr>
</tbody>
</table>

Fig. 5. Final software (left) and hardware (right) maps for the corner terrain with \(P = 500\), \(M = 9\), \(G = 25\), \(\text{ocgd} = 0.10\).
are partly due to the differences in implementation (e.g. finite word-lengths) and partly due to the random nature of the algorithm (two runs with the same parameter configuration never give exactly the same result).
The main problem of the algorithm is its tendency to accumulate rotational error as described in Section 2. This results in wrong loop closures and redrawing of the same obstacles in different positions. The enhancements proposed in Section 2, although helpful, do not eliminate rotational errors. Especially when the basic version of the system is used, the errors are intolerable. This can be seen in the simple corner terrain tests (Fig. 5), where the algorithm fails and cannot localize the robot correctly during the second corner crossing. The same problem appears in the small circle terrain (Fig. 7), where the loop closure is inaccurate. This error can be controlled by changing the parameter set. More specifically, an increase in the size of the population and a decrease of $M$ and $ocgd$ result in more accurate map drawing and less rotational error. This is particularly visible in the top version runs for the two terrains mentioned above (Figs. 6 and 8). The rotational distortion in the Rooms terrain map, which is more complex, is also decreased when the parameter set is improved (Figs. 9 and 10). The maps in Fig. 10 are closer to the actual shape of the terrain.

Table 5
Quantitative comparison (using MAE) of the hardware and software implementations for two sets of parameters.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Software MAE</th>
<th>Hardware MAE</th>
<th>Increase of MAE between software and hardware (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = 500, M = 9, G = 25, ocgd = 0.10$</td>
<td>5.5993</td>
<td>6.0108</td>
<td>7.3</td>
</tr>
<tr>
<td>$P = 2000, M = 5, G = 25, ocgd = 0.07$</td>
<td>2.9155</td>
<td>3.2115</td>
<td>10.1</td>
</tr>
</tbody>
</table>

Table 6
Speed results for the basic hardware version and comparison to software.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>$P = 500, G = 25, M = 9, ocgd = 0.07$</th>
<th>Theoretical number of cycles for one iteration (hardware)</th>
<th>Actual number of cycles for one iteration (hardware)</th>
<th>Actual time for one iteration (hardware)</th>
<th>Achieved speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical number</td>
<td>158,706 + X</td>
<td>160,463</td>
<td>1.1461 ms</td>
<td>16,2761 ms</td>
<td>14.20</td>
</tr>
<tr>
<td>Actual number of</td>
<td>972,722 + X</td>
<td>975,279</td>
<td>6.9662 ms</td>
<td>103,3472 ms</td>
<td></td>
</tr>
<tr>
<td>Actual time for</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7
Speed results for the top hardware version and comparison to software.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>$P = 2000, G = 25, M = 5, ocgd = 0.07$</th>
<th>Theoretical number of cycles for one iteration (hardware)</th>
<th>Actual number of cycles for one iteration (hardware)</th>
<th>Actual time for one iteration (hardware)</th>
<th>Achieved speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical number</td>
<td>158,706 + X</td>
<td>160,463</td>
<td>1.1461 ms</td>
<td>16,2761 ms</td>
<td>14.20</td>
</tr>
<tr>
<td>Actual number of</td>
<td>972,722 + X</td>
<td>975,279</td>
<td>6.9662 ms</td>
<td>103,3472 ms</td>
<td></td>
</tr>
<tr>
<td>Actual time for</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>one iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The main problem of the algorithm is its tendency to accumulate rotational error as described in Section 2. This results in wrong loop closures and redrawing of the same obstacles in different positions. The enhancements proposed in Section 2, although helpful, do not eliminate rotational errors. Especially when the basic version of the system is used, the errors are intolerable. This can be seen in the simple corner terrain tests (Fig. 5), where the algorithm fails and cannot localize the robot correctly during the second corner crossing. The same problem appears in the small circle terrain (Fig. 7), where the loop closure is inaccurate. This error can be controlled by changing the parameter set. More specifically, an increase in the size of the population and a decrease of $M$ and $ocgd$ result in more accurate map drawing and less rotational error. This is particularly visible in the top version runs for the two terrains mentioned above (Figs. 6 and 8). The rotational distortion in the Rooms terrain map, which is more complex, is also decreased when the parameter set is improved (Figs. 9 and 10). The maps in Fig. 10 are closer to the actual shape of the terrain.

In addition to the above qualitative comparison, a quantitative comparison of the accuracy of hardware and software outputs was performed. The generated maps were compared to the true maps by calculating the Mean Absolute Error (MAE) between corresponding cell values (the range of cell values is 0–255). The results for the Corner terrain are shown in Table 5. The increase in MAE between software and hardware is 7.3% for the first parameter set and 10.1% for the second parameter set. This increase is not practically significant and shows that the two implementations are comparable in terms of accuracy. The increase in MAE between the first parameter set and the second parameter set for both software and hardware is around 90% (which quantifies the distortion observed when comparing Figs. 5 and 6). Results for other terrains are similar.

Table 8
Speedup achieved in other hardware genetic algorithm implementation.

<table>
<thead>
<tr>
<th>Hardware implementation</th>
<th>FPGA</th>
<th>Test problem</th>
<th>Speedup over software</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6]</td>
<td>Xilinx XC2VP30</td>
<td>Various function maximizations</td>
<td>5.16×</td>
</tr>
<tr>
<td>[4]</td>
<td>Xilinx Spartan 3 1500-4FG676</td>
<td>TSP</td>
<td>11.035×</td>
</tr>
<tr>
<td>[13]</td>
<td>Xilinx Spartan 3</td>
<td>Maximization of Michalewic’s function</td>
<td>214.8×</td>
</tr>
<tr>
<td>[17]</td>
<td>Altera Cyclone (unspecified)</td>
<td>Knapsack problem</td>
<td>71.65×</td>
</tr>
<tr>
<td>[21]</td>
<td>Xilinx XCV2000E</td>
<td>TSP</td>
<td>38.627× (maximum)</td>
</tr>
</tbody>
</table>
more than 4000 genes, M values less than 5 and $ocgd$ values less than 0.05 should be used. This means increased computational load. In such cases, the hardware version will provide a valuable speed-up. On the other hand, it must be taken into account that the extent to which $ocgd$ can be decreased depends on the amount of block RAMs available on the target FPGA.

6.2. Speed

The total number of clock cycles needed for a complete iteration in the hardware implementation is the sum of clock cycles of the various modules:

$$C = P + \frac{180}{M} + G(\frac{180}{2M}P + P) + 181\frac{8}{ocgd} + X$$

where C is the total cycle number and X is a value representing the additional clock cycles needed for handshake between modules and some controller latency.

Tables 6 and 7 show the speed results of two versions of the hardware implementation (internal clock speed of 143 MHz), compared to their software equivalents. The software versions ran on a PC with Linux OS, Intel Core 2 Duo 3.2 GHz and 4 GB of RAM.

A speedup of up to 14.83× is achieved using the hardware system. This means that the hardware system can be used with an improved parameter set that provides increased accuracy. The SLAM task can be executed in the same time period that a software system with inferior accuracy would need. The fact that the FPGA is almost 15 times faster while operating at a clock frequency more than 20 times lower than the clock frequency of the CPU can be attributed entirely to pipelining and the use of parallel operators that were described in Section 4.

These results cannot be directly compared to results from other software/hardware SLAM implementations, since the algorithms implemented, the sensor input and the memory requirements are different. Having said that, the only complete FPGA implementation of a SLAM algorithm that was found [3] achieves an acceleration of the same order as the work presented here. It is three times faster than a desktop processor implementation and 13 times faster than an embedded processor implementation.

As for the hardware implementations of genetic algorithm cores, Table 8 lists the speedups documented for their respective test problems. Some works provide much higher speedups than the present one but this is to be expected, since they have considered only the implementation of the genetic core without any interaction with other components. Thus they can take advantage of the full parallelization potential of the genetic operations (by using more parallel computational blocks with the only limit being the FPGA’s resources). For example, in [13] all the genes of the population are updated simultaneously to form the new generation.

The implementation presented here accelerates the complete SLAM problem, one part of which is the genetic algorithm. It has to take into account the frequent access to the map memory, which places a limit on how much parallelism can be exploited. Nevertheless, even in this setting, the FPGA provides significant speedup, which demonstrates the generic suitability of a deeply pipelined architecture to the genetic algorithm task. Also, it is apparent that the flexibility of the FPGA's reconfigurable architecture is a great tool to the designer trying to accelerate an application, even when the application’s parallelism cannot be fully exploited.

7. Conclusion and future work

This work presents SMG-SLAM, a novel SLAM algorithm, and focuses on its hardware implementation using an FPGA. SMG-SLAM is a SLAM algorithm based on genetic algorithms and scan-matching and uses the measurements taken by an LRF to iteratively update a mobile robot's pose and map estimate. The hardware/software system designed exploited the inherent parallelism of the genetic algorithm and the fine-grain reconfigurability of the FPGA to achieve a speedup of up to 14.83× compared to the software version of the algorithm. The architecture is parameterized and can provide adaptable levels of accuracy. It was shown that the accuracy of the hardware implementation for the same set of parameters is comparable to that of the software implementation. The proposed hardware/software system can be integrated into a larger autonomous robotic system to perform SLAM efficiently.

This work can be extended in a number of ways. In order to improve the system's accuracy, the algorithm's structure can be enhanced taking into account measurements from additional sensors (e.g. ultra-sound) or measurements from a wider angle than 180°. Also, several elements of the hardware architecture (e.g. mutation strategy) can be implemented in different ways [4] to possibly achieve better results. To further accelerate the algorithm, two or more identical maps can be used in parallel in order to be able to access four or more map cells in each clock cycle. This would double the speed of the Fitness evaluation module (which is responsible for most of the computational time) but would require double the FPGA memory resources used normally. Furthermore, a modified version of the IGA architecture proposed in [17] can be employed to further parallelize the EMM module. The population and fitness memories would have to be split into several smaller memories which would be processed in parallel by the EMM module. Finally, a power consumption analysis can be performed in order to measure the power efficiency gains compared to a processor-based solution.

References


J. Wang, USARSim v.3.1.3 Tutorial: A Game-based Simulation of Mobile Robots, USARSim Sourceforge website 2004


Grigorios Mingas is a PhD student in the Department of Electrical and Electronic Engineering, Imperial College London, UK. His research interests include reconfigurable computing and Markov Chain Monte Carlo simulation.

Emmanouil Tsardoulias is a PhD student in the Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, Greece. His research focuses on the investigation of SLAM, planning, navigation and communication in a group of rescue robots using genetic methods and RFID.

Loukas Petrou is an Associate Professor in the Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, Greece. His research interests include microprocessor applications, autonomous mobile robots, intelligent control of waste water treatment plants, reconfigurable embedded systems, evolutionary computation, fuzzy logic and neural network control systems.