SIMD and OpenMP Optimization of EKF-SLAM

Bastien Vincke, Abdelhafid Elouardi, Alain Lambert and Abdelhamid Dine

***Abstract*—SLAM algorithms are widely used by autonomous robots operating in unknown environments. Several works have presented optimizations mainly focused on the algorithm complexity. New computing technologies (SIMD coprocessors, multi-core architecture) can greatly accelerate the processing time but require rethinking the algorithm implementation. This paper presents an efficient implementation of the EKF-SLAM algorithm on an embedded system implementing OMAP multi-core architecture and SIMD optimizations. The aim is to optimize the algorithm implementation to improve the localization quality. Results demonstrate that an optimized implementation is always needed to achieve efficient performances and can help to design embedded systems implementing a low-cost multi-core architecture operating under real time constraints.**

Keywords: EKF-SLAM, multi-core implementation, OMAP architecture.

I. Introduction

Many researches were made to develop SLAM (Simultaneous Localization And Mapping) algorithms to build an environment map while estimating the robot pose like EKF-SLAM (Extended Kalman Filter for Simultaneous Localization And Mapping) [2], FastSLAM or GraphSLAM. Nevertheless, these algorithmes are rarely implemented on low-cost architecture based embedded systems because classical SLAM algorithms are too computationally intensive to run on an embedded computing unit.

SLAM algorithm traditionnaly require at least laptop-level performances. Some embedded implementation were proposed as [3] presents a low-cost approach to autonomous multi-robot mapping and exploration for unstructured environments. The algorithm runs on a Gumstix computing unit (600 MHz) and the robot embbeds 6 IR scanning range arrays, a 3-axis gyroscope and odometers. Each updating step takes on average 3 seconds with 15 particles and more than 10 seconds with 25 particles. An other system was proposed by [5]. The system is based on the co-design of a low-cost sensor (a slim rotating scanner), a SLAM algorithm, a computing unit, and an optimization methodology. The computing unit is based on an ARM processor (533 MHz) running a FASTSLAM 2.0 algorithm. Moreover, [5] uses an evolution strategy to find the best configuration of the algorithm and setting of the parameters. As pointed out by [5] and [3], the first improvement of a SLAM algorithm is an efficient setting of the various algorithm parameters. Other modifications were investigated to reach real time constraints. These modifications are necessary due to the low computing power and limited memory resources available on embedded systems.

Previously, we improve on previous works by proposing an adequate implementation of the EKF-SLAM algorithm on a multi-core architecture [8]. We proposed a SLAM system based on a EKF-SLAM algorithm and a OMAP3530 ARM architecture. The robot embbeds a low-cost camera as an exteroceptive sensor and two odometers as proprioceptive sensors. Our design methodology was explained and the results show that it is possible to design real time SLAM system despite it could not handle a large set of landmarks. More recently, we proposed a new evaluation of our custom algorithm on a OMAP4430 architecture [9] without implementing a fast matrix multiplication. The comparison between our two systems shows an improvement.

This paper presents an efficient implementation of the estimation step of the EKF-SLAM algorithm on a multi-core ARM architecture (OMAP4430). The approach is based on an algorithm-architecture adequacy which consists to optimize the implementation of the matrix multiplication step on a low-cost architecture implementing a Dual-core ARM Cortex A9 integrating two SIMD NEON coprocessors. The specifications related to the NEON coprocessors and the multi-core processing improve the computing time and the system performance. Using an optimized algorithms and an adequate architecture, it is possible to implement a SLAM algorithm on a low-cost multi-core architecture operating under real time constraints.

Section 2 introduces the EKF-SLAM algorithm. Section 3 presents the embedded multi-core architecture and the system configuration. Section 4 details the block partitioning of the algorithm and analyzes a first implementation in terms of processing time. A software optimization is proposed and analyzed in Section 5. It presents SIMD optimizations and multi-core parallelization. A performance comparison is then performed between the optimized and non-optimized instances. Finally, section 6 concludes this paper.

II. EKF-SLAM Algorithm

A. Overview

EKF-SLAM estimates a state vector containing both the robot pose and the landmarks location. It uses proprioceptive sensors to compute a predicted vector and then corrects this state using exteroceptive sensors. In this paper, we consider a wheeled robot embedding two odometers (attached to each rear wheel) and a camera.

1) State Vector and Covariance Matrix: With N landmarks, the state vector is defined as:

$$\mathbf{x} = (x, z, \theta, x_{a_1}, y_{a_1}, z_{a_1}, ..., x_{a_N}, y_{a_N}, z_{a_N})^T \qquad (1)$$

where :

- x, z are the ground coordinates (x-axis, z-axis) of the robot rear axle center and θ is the orientation of a local frame attached to the robot with respect to the global frame.

Univ Paris-Sud, CNRS, Institut d'Electronique Fondamentale, F-91405 Orsay, France. bastien.vincke@u-psud.fr

978-1-4799-3824-7/14/$31.00 ©2014 IEEE

- $x_{a_1}, y_{a_1}, z_{a_1}, ..., x_{a_N}, y_{a_N}, z_{a_N}$ are the 3D coordinates of the N landmarks in the global frame.

2) Prediction : The prediction step relies on the measurements of the proprioceptive sensors, the odometers, embedded on our experimental platform. A non linear discrete-time state-space model is considered to describe the evolution of the robot configuration $\mathbf{x}$:

$$\mathbf{x}_{k|k-1} = \mathbf{f}(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k) + \mathbf{v}_k \tag{2}$$

where $\mathbf{u}_k$ is a known two-dimensional control vector, assumed constant between the times indexed by $k-1$ and k, and $\mathbf{v}_k$ is an unknown state perturbation vector that accounts for the model uncertainties. During the prediction process, the landmarks localization do not change. The classical evolution model, described in [7], is considered:

$$\mathbf{f}(\mathbf{x}_{k-1|k-1}, \delta s, \delta\theta) = \begin{pmatrix} x_{k-1} + \delta s \cos\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ z_{k-1} + \delta s \sin\left(\theta_{k-1} + \frac{\delta\theta}{2}\right) \\ \theta_{k-1} + \delta\theta \end{pmatrix} \tag{3}$$

where $\mathbf{u}_k = (\delta s, \delta\theta)$; δs is the longitudinal motion and $\delta\theta$ is the rotational motion.

3) Estimation : The estimation of the state is made using the camera which returns the position in the image (u_i, v_i) of the i-th landmark. The pinhole model is used to predict the observation for a single landmark:

$$\mathbf{h}_i(\mathbf{x}_{k|k-1}) = \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} c_u + fk_u \frac{x_{a_i}^{cam}}{z_{a_i}^{cam}} \\ c_v + fk_v \frac{y_{a_i}^{cam}}{z_{a_i}^{cam}} \end{pmatrix} \tag{4}$$

where (u_i, v_i) is the position of the i-th landmark in the image, $(x_{a_i}^{cam}, y_{a_i}^{cam}, z_{a_i}^{cam})$ is the position of the i-th landmark in the camera frame, f is the focal length and (k_u, k_v) is the number of pixels per unit length. During the observation step, the algorithm matches M landmarks ($M <= N$) whose observations are added in $\mathbf{h} = \begin{pmatrix} h_1, h_2, .., h_M \end{pmatrix}^T$. Then, the state is updated using the classical well-known EKF equations which mainly consist in matrix multiplication.

B. Landmarks Detection, Initialization, Matching and Map management

The landmarks used in the observation equation are extracted from images. We select FAST detector [6] as the landmark detector. This algorithm is commonly used by SLAM algorithm.

Landmark initialization consists in defining the initial coordinates and the initial covariance of landmarks (interest points). We choose to use the widely spread delayed method proposed by [2] which is both efficient and adequate to implement.

For the matching, the EKF-SLAM matches the previously detected feature with a new one using Zero-Mean Sum of Squared Differences (ZMSSD). We use the ZMSSD to find the best candidate point. The ZMSSD is computed as:

$$\begin{aligned} N_p \quad = \quad & (((2Sd \times Si) - Sd^2 - Si^2)/256) \\ & + SSi + SSd - 2Sdi \end{aligned} \tag{5}$$

Where:

- $Sd = \sum_{i,j} d(i,j)$ the sum of the descriptor pixels (this sum can be precalculated).
- $Si = \sum_{i,j} im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ the sum of the image pixels.
- $SSi = \sum_{i,j} im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}) \times im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ the sum of image squared pixels.
- $SSd = \sum_{i,j} d(i,j) \times d(i,j)$ the sum of the descriptor squared pixels (this sum can be precalculated).
- $Sdi = \sum_{i,j} d(i,j) \times im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2})$ the sum of the product of the pixels descriptor and the image (this sum can be precalculated).

The des parameter is the descriptor size. The observation will be selected by minimizing the ZMSSD between all candidated. The descriptor, used to identify the landmark during the matching, is classically a small image window of 9×9 pixels to 16×16 pixels around the interest point.

To keep the size of map constant, we need to delete some landmarks when inserting new ones. [1] proposes an efficient method to select landmarks for the estimation task. It is based on the evaluation of the influence of a given feature on the convergence of the state covariance matrix. The method matches all possible landmarks and computes $(\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)$ from the Kalman estimation equations.

III. System Architecture

Our embedded system is based on the OMAP4430 multi-core architecture: two ARM Cortex A9 (2*1 GHz), 2*SIMD NEON coprocessors and one GPU PowerVR. Multiple sensors are interfaced to this architecture: odometers (HEDS 5540) and a vision sensor (Kinect). A coprocessor (ATMega168) is also implemented to control the robot speed and its direction using two PWM (Pulse-Width Modulation). It decodes signals coming from the odometers embedded on each wheel. It communicates with the main board using an I2C interface. This interface allows the main processor to retrieve odometers data and to send instructions corresponding to speed and direction.

During the experimentation, frames have been grabbed at 30 fps with 640×480 resolution. Odometers data were sampled at 30 Hz.

IV. Functional Block Partitioning

The evaluation methodology consists to identify the processing tasks requiring a meaning computing time. It is based on several steps: we analyze first the execution time of tasks and their dependencies on the algorithm's parameters. The algorithm is then partitioned in order to have functional blocks (FBs) performing defined calculations. Each block is then evaluated to determine its processing time. Function blocks that require the most important execution time are then optimized in adequacy to the based architecture to reduce the overall

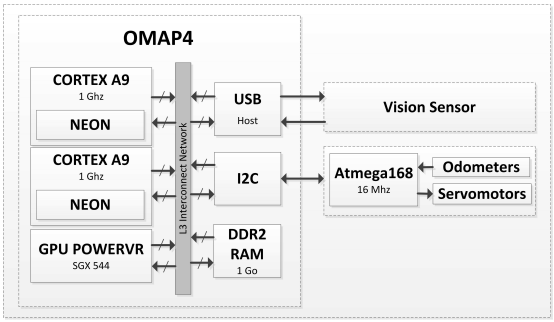

Figure 1. System architecture

processing time. EKF-SLAM is composed of two process: Prediction and Correction. The correction process implements three tasks: matching, estimation and initialization.

EKF-SLAM tasks do not have a fixed computing time, this time depends on the experiments (number of landmarks, robot and landmarks localization uncertainties). We have to define Functional Blocks (FB) which have a fixed computing time. The tasks must be independent in term of FBs processing times. Experiments will only impact the number of occurrences of the FBs. The matching tasks has been divided into 3 FBs: landmarks projection, ZMSSD for Matching (ZMSSD-M) and $\mathbf{H}_i$ computation. The initialization task has been divided into 3 FBs: ZMSSD for Initialization (ZMSSD-I), Weight updating and Addition of a new landmark. Prediction and Estimation have not been divided because they have a constant processing time. To summarize, 9 FBs have been defined:

1) Prediction: the entire prediction process.
2) FAST: the FAST detector application.
3) Landmark projection: the projection of one landmark on the camera plane.
4) ZMSSD-M : the correlation computation between one candidate point of the image and one descriptor for the matching task.
5) $\mathbf{H}_i$: $\mathbf{H}$ computation for the i-th observation.
6) Estimation: the entire estimation task.
7) ZMSSD-I : the correlation computation between one candidate point of the image and one descriptor for the initialization task.
8) Weight updating: the update of the particle weight for the initialization step.
9) Addition of a new landmark: the insertion of a new landmark under initialization.

Each FB has a fixed computing time and some FBs can occur more than one time (Landmark projection, ZMSSD-M, ZMSSD-I, $\mathbf{H}_i$, Weight updating, Addition of a new landmark).

A. Processing Time Evaluation

As an application scenario, the robot moves over a square of 6 meters side. At the end of the trajectory, it joined the initial starting position. Using only odometers, the final localization has an error of 1.6 m. With the EKF-SLAM algorithm, the localization has been significantly improved. The final error is approximately 0.4 m. EKF-SLAM includes all viewed landmarks in the state vector. Indeed, the localization result depends on the number of landmarks but the size of the state vector and the number of observations must be bounded to achieve a bounded computing time. The overall accuracy of the EKF-SLAM depends on the number of the landmarks in the state vector and the matched observations. More the number of processed landmarks is important, more the localization is accurate. We set the size of the descriptor to 16x16 pixels, the maximum number of landmarks in the state vector to 50, the maximum number of observed landmarks to 50 and the maximum number of landmarks being initialized to 50. First, we can analyze the runtime of the 9 previously defined Functional blocks (FBs) of the algorithm. The prediction process (FB1) occurs in 0.026 ms. Table I summarizes, for the FBs, the processing time per occurrence, the mean number of occurrences and the mean processing time per correction process. Estimation FB does not occur at each frame. It only occurs if there is at least one matched landmark. The mean processing time by frame is approximately 65.90 ms which corresponds to the sum of all processing times: prediction process (FB1) and correction process (FB2 to FB9). The processing time of the estimation task (FB6) is approximately 55.1 ms and it represents about 83% of the global processing time. The FAST detector (FB2) occurs in 6.5 ms. ZMSSD-M (FB4) takes 3.23 ms per correction process. Finally, the initialization tasks (FB7 to FB9) takes 0.8 ms. These FBs (2, 4, 6 and 7) represent 98.6% of the global processing time. We will focus on an efficient implementation of these FBs to enhance the global processing time.

Table I
FBs Processing time in the correction process on a single ARM core

Functional Block (FB)	Processing time per occurrence (μs)	Mean of the number of occurrences per correction process	Mean of the processing time per correction process (μs)
2. FAST	6581	1	6581
3. Landmark projection	1.37	40.9	56.03
4. ZMSSD-M	3.46	933.7	3230.60
5. $\mathbf{H}_i$	3.42	15.86	54.24
6. Estimation	61317	0.90	55185.30
7. ZMSSD-I	3.46	158.03	546.78
8. Weight updating	47.13	5.07	238.94
9. Addition of a new landmark	32.18	0.26	8.36
Total	**-**	**-**	**65901.25**

V. Optimization and Improvements

A. OMAP4430 Architecture Description

The OMAP4430 is an architecture designed by TI (Texas Instruments) and implements a dual core ARM Cortex-A9 1 GHz, each core has a NEON coprocessor with SIMD instructions. NEON is optimized for Single Instruction Multiple Data (SIMD) operations. NEON instructions perform "Packed SIMD" processing. The registers are considered as vectors of the same data elements. Data types can be: signed/unsigned

8-bits, 16-bits, 32-bits, 64-bits or single precision floating point. The instructions perform the same operation on multiple data simultaneously. The number of simultaneous operations depends on the data type: NEON supports up to 16 operations at the same time using 8-bits data.

B. SIMD and Multi-core Optimization Results

The time-consuming functional blocks are: the estimation (FB6), FAST detector (FB2) and the ZMSSD (FB4, FB7). FAST corner detector is already an optimized instance using machine learning but it can be parallelized using the two cores of the OMAP4430. The matching and initialization tasks compute ZMSSD which consists in computing image correlation. It performs the same operation on 8 bits data. The computation of the ZMSSD can be optimized using SIMD NEON instructions. The estimation step is based on floating point matrix multiplication, it could efficiently be optimized using the NEON coprocessor.

a) FAST Detector Implementation: FAST detector [6] relies on a simple test performed for each pixel. To implement the FAST detector (FB1) on the two available ARM cores, we choose to horizontally slip the image into two overlapping sub-images (6 pixels overlapping, due to the test perform on a Bresenham circle of radius 3). FAST is processed on the two sub images and the results are concatenated. The FAST algorithm, implemented on a single ARM core, takes 6581 μs. The Dual-core implementation takes 3701 μs. The optimized implementation represents 56% of the single-core implementation.

b) ZMSSD SIMD Optimization: The EKF-SLAM matches features using Zero-Mean Sum of Squared Differences (ZMSSD). ZMSSD is computed for each landmark using Eq. (5). SIMD NEON architecture allows vector processing and performs the same operation on all the vector processing-units. We have implemented a vectorized instance of the ZMSSD function as follows:

Algorithm 1 SIMD Vectorized ZMSSD

1: $V8x8\ Vimage \leftarrow 0$ ▷ V8x8: Defines a 8×8 bits vector
2: $V8x8\ Vdescriptor \leftarrow 0$
3: $V16x8\ VSi \leftarrow 0$ ▷ V16x8: Defines a 8×16 bits vector
4: $V32x4\ VSSi \leftarrow 0$ ▷ V32x4: Defines a 4×32 bits vector
5: $V32x4\ VSdi \leftarrow 0$
6: **for Each** $i \in [0;15]$, $j = 0, 8$ **do**
7: $Vimage \leftarrow load_8(im(p_x + i - \frac{des}{2}, p_y + j - \frac{des}{2}))$ ▷ Load 8 pixels
8: $Vdescriptor \leftarrow load_8(d(i,j))$
9: $VSi \leftarrow VSi + Vimage$
10: $VSSi \leftarrow VSSi + Vimage \times Vimage$
11: $VSdi \leftarrow VSdi + Vimage \times Vdescriptor$
12: **end for**
13: $Si \leftarrow sum(VSi)$ ▷ Sums the component of a vector
14: $SSi \leftarrow sum(VSSi)$
15: $Sdi \leftarrow sum(VSdi)$
16: $Np \leftarrow (((2Sd \times Si) - Sd^2 - Si^2)/256) + SSi + SSd - 2Sdi$

This instance uses 8 pixels at time. SIMD NEON architecture allows computing 8 addition or 8 multiplication simultaneously. The processing time of the vector implementation decreases to 0.58 μs (6 times faster than the ARM implementation taking 3.46 μs).

c) Matching and Initialization Parallelization: To efficiently optimize the matching and initialization tasks on the Dual-core architecture, we choose to parallelize the entire tasks. We use OpenMP tools to parallelize the loop execution over the landmarks. Without any optimization (using a single ARM core), the entire matching task processing time is 3340.87 μs (for FB3, FB4 and FB5). Using SIMD NEON coprocessor with ARM core, these computing time decreases to 670.35 μs. Finally, with the Dual-core optimization (Dual-ARM and Dual-NEON), the processing time decreases to 454.51 μs. For the initialization, the ARM processing time is 794.08 μs. It is reduced to 338.97 μs using the SIMD NEON coprocessor and ARM core. This time is reduced to 225.6 μs with OpenMP parallelization (implementing Dual-ARM and Dual-NEON).

d) Estimation Optimization : Matrix multiplication optimization is a classical problem that has been already studied. The embedded system implements 2 SIMD NEON coprocessors. In our knowledge, there is only one library which includes an optimized matrix multiplication for ARM SIMD Multi-core architecture: EIGEN3 library. This library implements the optimization proposed by [4].

We focus on the case of a squared matrix multiplication of size N. Matrix is stored using column-major order. Matrix multiplication ($C = A \times B$) is classically processed in 3 loops:

for $i \in [0; N]$
  for $j \in [0; N]$
    for $k \in [0; N]$
      $C(i,j) += A(i,k) \times B(k,j)$

To implement SIMD instructions, each matrix is divided into 4*4 blocks because SIMD instructions can deal with 128 bits vectors (4 × 32bits vectors). The matrix multiplication algorithm becomes:

for $i \in [0 : 4 : N]$
  for $j \in [0 : 4 : N]$
    for $k \in [0 : 4 : N]$
      $C(i : i+3, j : j+3) += A(i : i+3, k : k+3) \times B(k : k+3, j : j+3)$

Matrix size must be a multiple of 4.In [4], authors propose an optimization which consists in preloading rows of the matrix A into the cache memory. This optimization reduces the access to the DDR memory and it performs linear access in the cache memory. This optimization is included in the Eigen library. We propose an new optimization based on the SIMD NEON architecture. This architecture includes 16 registers of $128bits$. Classical optimization uses only 12 vectors: 4 vectors for each matrix A, B and C. In this case, the load/store instructions are time-consuming operations. In order to limit the number of the load/store instructions, we propose to implement the 16 available vectors. The algorithm computes simultaneously two blocks of the C matrix and becomes:

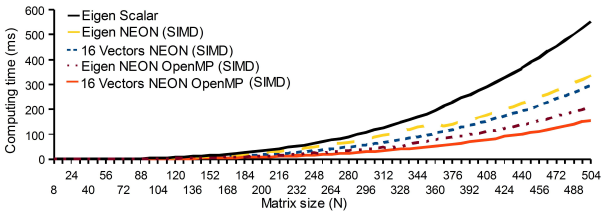

Figure 2. Processing time of the matrix multiplication on ARM and NEON coprocessor.

Table II
TASK'S PROCESSING TIME ON SINGLE AND MULTI-CORE ARCHITECTURE.

Task	FB	Mean processing time per Frame (μs)	
		Single Core Implementation	Multi-Core Implementation
FAST	2	6581	3701
Matching	3, 4, 5	3340.87	454.51
Estimation	6	55185.30	19584.26
Initialization	7, 8, 9	794.08	225.6
Total		**65901.25**	**23965.37**

for $i \in [0:8:N]$
  for $j \in [0:4:N]$
    for $k \in [0:4:N]$
      $C(i:i+3, j:j+4) += A(i:i+4, k:k+4) \times B(k:k+4, j:j+4)$
      $C(i+4:i+7, j:j+4) += A(i+4:i+7, k:k+4) \times B(k:k+4, j:j+4)$

We efficiently use the 16 vectors: 8 vectors for the C matrix, 4 vectors for each matrix A and B. Moreover, there is no need to reload $B(k:k+4, j:j+4)$ sub-matrix or to store the 8 vectors of the C matrix. The multi-core implementation consists to parallelize the $i^{th} - loop$ on the two NEON coprocessors. Due to the embedded constraints, we have evaluated the matrix multiplication for different sizes (8 to 512). Figure 2 represents the obtained results. It shows that our implementation is always faster than those of Eigen. The processing time of the matrix multiplication (Dual NEON-core) is approximately 3.5 times faster than those of the Eigen scalar one (implementing a single core).

C. Global Results

We have improved the EKF SLAM implementation using two principal specifications of the based system: the SIMD NEON coprocessors and the Dual-core architecture. Table II summarizes the processing time of the different tasks. We cannot use the same description as Table I because some tasks have been parallelized (Matching and Initialization tasks). The mean processing time per frame with the optimized implementation is 23.9 ms whereas the single core implementation has a processing time of 65.9 ms. The optimized processing time represents 36% of the single-core processing-time. In comparison with our previous implementation which did not implement the optimized matrix multiplication [9], we have reduced by 10% the mean processing time.

VI. CONCLUSION

This paper proposed an efficient implementation of the EKF-SLAM algorithm on a multi-core architecture. Based on the application constraints, we have implemented the algorithm in adequacy with the based architecture. A runtime analyzes shows that 4 FBs represents more than 98% of the global processing time. We have parallelized the FAST functional block on the Dual-core architecture. Three tasks (matching, initialization and estimation) have been optimized using the SIMD NEON architecture and have been parallelized on the multi-core architecture. Using the optimized implementation, the global processing time was reduced by a factor of 2.75. The results demonstrate that an optimized implementation, can help to design embedded systems implementing a low-cost multi-core architecture operating under real time constraints.

REFERENCES

[1] F.A. Auat Cheein and R. Carelli. Analysis of different feature selection criteria based on a covariance convergence perspective for a slam algorithm. *Sensors*, 11(1):62–89, 2010.

[2] A.J. Davison, I.D. Reid, N.D. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *IEEE Trans on Pattern Analysis and Machine Intelligence*, pages 1052–1067, 2007.

[3] C.M. Gifford, R. Webb, J. Bley, D. Leung, M. Calnon, J. Makarewicz, B. Banz, and A. Agah. Low-cost multi-robot exploration and mapping. In *IEEE Int. Conf. on Technologies for Practical Robot Applications*, pages 74–79, 2008.

[4] K. Goto and R.A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12, 2008.

[5] S. Magnenat, V. Longchamp, M. Bonani, P. Rétornaz, P. Germano, H. Bleuler, and F. Mondada. Affordable slam through the co-design of hardware and methodology. In *IEEE Int. Conf. on Robotics and Automation*, pages 5395–5401, 2010.

[6] 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.

[7] 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.

[8] Bastien Vincke, Abdelhafid Elouardi, and Alain Lambert. Real time simultaneous localization and mapping: towards low-cost multiprocessor embedded systems. *EURASIP Journal on Embedded Systems*, 2012.

[9] Bastien Vincke, Abdelhafid Elouardi, Alain Lambert, and Alain Merigot. Efficient implementation of ekf-slam on a multi-core embedded system. In *Industrial Electronics Society IECON*, pages 3049–3054, 2012.