Abouzahir *et al. EURASIP Journal on Advances in Signal Processing* (2016) 2016:88
DOI 10.1186/s13634-016-0386-3

EURASIP Journal on Advances in Signal Processing

RESEARCH Open Access

Large-scale monocular FastSLAM2.0 acceleration on an embedded heterogeneous architecture

Mohamed Abouzahir[1*], Abdelhafid Elouardi[1], Samir Bouaziz[1], Rachid Latif[2] and Abdelouahed Tajer[3]

Abstract

Simultaneous localization and mapping (SLAM) is widely used in many robotic applications and autonomous navigation. This paper presents a study of FastSLAM2.0 computational complexity based on a monocular vision system. The algorithm is intended to operate with many particles in a large-scale environment. FastSLAM2.0 was partitioned into functional blocks allowing a hardware software matching on a CPU-GPGPU-based SoC architecture. Performances in terms of processing time and localization accuracy were evaluated using a real indoor dataset. Results demonstrate that an optimized and efficient CPU-GPGPU partitioning allows performing accurate localization results and high-speed execution of a monocular FastSLAM2.0-based embedded system operating under real-time constraints.

Keywords: Monocular FastSLAM2.0, CPU, GPGPU, Heterogeneous architecture, Hardware software matching

1 Introduction

Simultaneous localization and mapping (SLAM) algorithms are computationally intensive. Therefore, there is a general need, in case of embedded systems, to have an architecture that allows a software optimization for efficient and scalable implementation. Computer systems, in the early days of their creation, have contained one kind of processors designed to run general computing tasks. Performance improvement of such computers was relied to Moore's law which predicts doubling transistor density every 18 months. However, this trend has reached a certain maturity. It is no longer possible to gain performance by increasing transistor density because adding more transistors also adds high complexity, heat, and memory issues. In order to surpass these issues and to reach high performances, a new trend now is to include other processing elements in a single chip area. These new systems gain performance not by adding only the same type of processing units but implementing also dissimilar processors incorporating specialized capabilities dedicated for handling specific tasks. These systems are referred to as heterogeneous system architectures (HSAs). Such systems allow development of applications that seamlessly integrate CPUs with the most prevalent processing elements: GPUs. Heterogeneous architectures of such systems can be exploited to accelerate execution time of SLAM algorithms and make them operating in real-time constraints.

This article presents an algorithm architecture matching of FastSLAM2.0 algorithm on a heterogeneous architecture integrating a CPU and a GPGPU. Only few works deal with the implementation of FastSLAM2.0 on such embedded systems. Authors in [1] presented a hardware software co-design approach of the importance weight calculation and particle update on a NIOS II processor. Moyers et al. [2] presented a fixed-point version of FastSLAM 2.0 algorithm and describes its implementation on a configurable and extensible very long instruction word (VLIW) processor. Chau et al. [3] presented a heterogeneous implementation of adaptive particle filters based on an field-programmable gate array (FPGA) and a CPU for mobile robot localization and people tracking applications. There is no full implementation of FastSLAM2.0 on an embedded CPU-GPGPU published to this date. In the state of the art, we found that only the particle filter, as an algorithm of its own used for filtering, has recently been studied. Tosun et al. [4] presents a parallelization of particle filter-based localization and mapping on a multi-core

*Correspondence: mohamed.abouzahir@u-psud.fr
[1] Institut d'Electronique Fondamentale, Université Paris-Sud, 91405 Orsay, France
Full list of author information is available at the end of the article

architecture. Maskell et al. [5] presents a parallel resampling on an FPGA and [6] presents a GPU implementation of a general particle filter algorithm.

The work presented in [7] proposed a method to accelerate robot localization and mapping of FastSLAM1.0 algorithm. Authors exploit general purpose parallel computing on NVIDIA GPUs. However, this work only focuses on accelerating one task of the entire FastSLAM1.0 algorithm: the particle weight computation as a part of the resampling step on GPU where the other tasks of FastSLAM1.0 are executed on CPU. They also optimized memory access using textures. Furthermore, they evaluated their implementation on a desktop machine using a high-end GPU (NVIDIA GeForce GTX 660) and a CPU (Intel Core i5-3570K). Their tests have shown significant performance improvements according to the naive implementation on CPU.

In contrast, the purpose of our contribution is to present an efficient partitioning of a monocular FastSLAM2.0 on a heterogeneous embedded architecture and to provide the first complete GPGPU implementation of the algorithm with a focus on *general purpose processing unite*. The main contributions are

- The algorithm implemented in our work is slightly different from literature. Eade and Drummond [8] used only a single camera; the particle poses are predicted from images via visual odometry using a specific camera motion model. They tested the algorithm on a short sequence with few images for small environments. They claimed that the implemented algorithm is not yet ready for use in large-scale environments and significant challenges must be adopted. In our work, we used odometry to predict the particles poses and a single monocular camera for vision task. Some improvements are adopted in order to test the algorithm in large-scale environments. More details are given in Section 2.
- Our work proposes a full parallel implementation of FastSLAM2.0 on a GPU. Such implementation has not been reported before in any previous work. All main steps of the algorithm (functional blocks) have been accelerated on a GPU using the GPGPU concept. Zhang and Martin [7] accelerated only the particle weight calculation. This was a challenge to map the entire algorithm on a GPU.
- Zhang and Martin [7] used a high-end NVIDIA GPU of a desktop machine. Our work investigated a parallel GPGPU implementation of a heavy computationally algorithm (monocular FastSLAM2.0) on an embedded architecture. This rises a new challenge to investigate whether by adopting the same optimization strategies as those used for high-end GPU is suitable to design an embedded system-based SLAM applications. This challenge is due to constraints of the used architecture-based system on chip (NVIDIA Tegra K1 SoC) such as sharing the same physical memory between CPU and GPU. This required a special attention and imposed new demands to our works.

Organization of this paper is as follows: in Section 2, a state of the art about SLAM algorithms is presented with a description of image processing used for features extraction and matching. In the same section, the monocular FastSLAM2.0 algorithm is presented. Section 3 presents our methodology adopted in this work to evaluate the algorithm on a heterogeneous embedded architecture. In Section 4, a description of the target embedded architecture is provided with a brief background about hardware and material used for embedded GPGPU programming. The GPGPU implementation as well as the adopted partitioning model and the performed optimization is given in the same section. Section 5 is devoted to experimental results and provides a detailed discussion about the algorithm complexity and performances comparison. Section 6 summarizes results and gives a conclusion about this work.

2 Localization and mapping

SLAM algorithms allow autonomous navigation of robots in unknown environments. Localization and mapping represent a concurrent problem that cannot be solved independently. Indeed, if a mobile robot follows an unknown trajectory in an unknown environment, the estimation of the robot's pose and the explored map becomes more complicated. In such situation, no information is previously known by the mobile robot which is supposed to create a map and to localize itself according to this map. Before the robot can estimate the position of a given landmark, it needs to know from which location this landmark was observed. At the same time, it is difficult to estimate the actual position of the robot without a map. A good map is necessary for localization while an accurate pose estimate is needed for map reconstruction.

A SLAM algorithm relies on sensor data to concurrently estimate both map and robot pose. Two sensors are usually used: proprioceptive and exteroceptive sensors. Throughout this work, we used a monocular camera as an exteroceptive sensor to observe environment and odometers as proprioceptive sensors to estimate robot pose.

2.1 Image processing

Monocular SLAM algorithms use visual landmarks extracted from images to map the environment and to improve the robot localization. Landmarks are extracted using feature detectors. In our implementation, we used

FAST detector (features from accelerated segment test) [9]. It is less time consuming and suitable for real-time applications. FAST corner detector uses a circle of 16 pixels (radius = 3 pixels) to classify whether a candidate point p is actually a corner (observation in the context of mapping). If a set of n ($n = 9$, 10, 11 or 12) contiguous pixels in the circle are all brighter than the intensity of the candidate pixel p (denoted by I_p) plus a threshold value t ($I_p + t$) or all darker than the intensity of the candidate pixel p minus the threshold value t ($I_p - t$), then p is classified as a corner (Fig. 1).

SLAM systems need to detect previously observed landmarks in a reliable and robust way. Therefore, the extracted landmarks are identified by a simple descriptor which consists of a window of pixels around the observed landmark. Matching between observation and landmarks is achieved using a correlation-based similarity measure. We used the zero mean sum of squared differences (ZMSSD) metric in order to compute the similarity between descriptors of both landmarks (lmk) and extracted features. Equation 1 defines the ZMSSD formula.

$$\sum_{i,j} \left(I_{\text{lmk}}(i,j) - m_d - I_p(x+i, y+j) + m_p\right)^2 \tag{1}$$

2.2 FastSLAM2.0

FastSLAM2.0 [10] is based on the particle filter (Algorithm 3) . Uncertainty of the robot pose is modeled by a number of different particles. Each particle has its own map. It has been proved that FastSLAM2.0 runs successfully in a very large environment and can surpass many problems that decrease consistency of localization and mapping. The main steps of the algorithm are described below:

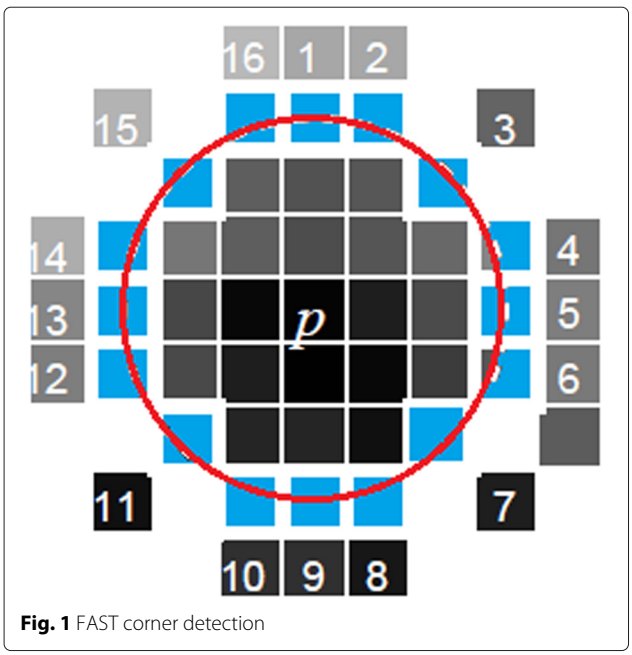

Fig. 1 FAST corner detection

2.2.1 Prediction

Prediction task propagates the current state of particles in the filter using motion model (2). The model incorporates odometer data u_t (n_l, n_r), where n_l and n_r are respectively the left and right wheel encoder data [11]. s_t (s_x, s_y, s_θ) is the particle pose, $\delta_s = \frac{n_l+n_r}{2}$ and $\delta_\theta = \frac{n_l-n_r}{2b}$ are respectively longitudinal and angular displacements, and b is the wheel base.

$$f(s_{t-1}, u_t) = s_t^m = s_{t-1}^m + \begin{pmatrix} \delta_s \cos\left(s_\theta + \frac{\delta_\theta}{2}\right) \\ \delta_s \sin\left(s_\theta + \frac{\delta_\theta}{2}\right) \\ \delta_\theta \end{pmatrix} \tag{2}$$

Prediction task also predicts uncertainty of the robot pose. We recall that as stated in [10], the difference between the FastSLAM2.0 and the previous version is how actually the particle poses are sampled. As we will see in Section 2.2.2, the particle poses are updated and sampled from a modified proposal distribution constructed incrementally using the last observed landmarks. The problem here is that such proposal distribution is constructed starting from an estimation of robot uncertainty, P_m. A random initialization of this uncertainty can cause the filter divergence.

Eade and Drummond [8] and Montemerlo et al. [10] did not give any information about how this initial covariance matrix is computed. Initializing P_m using an empirical value is not suitable when using a camera and partial initialization method since an accurate estimate of uncertainty is needed. This is shown in our previous work [12]: "Using small number of particles with P_m randomly initialized, the algorithm can only map a small portion of the trajectory and is not able to close the loop." In this work, we computed P_m incrementally whenever a new particle pose is predicted to accumulate uncertainty between images. This better reflects the uncertainty in robot pose for a good partial initialization. P_m must be initialized after each image acquisition.

Our work targets an embedded platform. The method we suggested is time consuming when it is naively implemented on an embedded single core, since P_m must be computed for every single particle and at every odometry data acquisition. An efficient optimization will be proposed in Section 4. Algorithm 1 describes the procedure of computing particle poses and their initial covariance matrix P_m according to one odometric data acquisition.

Density of particles must well represent the real trajectory (after integrating odometric data, particle density must cover the real robot pose). Therefore, in the prediction task, the motion model must be applied with a

randomization to reflect the system random error and the sensor noise (Monte Carlo localization) [13]. In our implementation, the prediction task is parallelized on GPU. However, the random number generation is difficult to be implemented in parallel on a GPU. According to [14], a random number generation should perform well on a single processor. Thus, providing high precision random numbers becomes more difficult when dealing with parallel architectures. Some studies [15, 16] proposed well-known techniques to spread random streams through a parallel implementation using different strategies, but they are not yet suitable for use in Monte Carlo localization and remain an ongoing research topic [14, 17, 18]. A solution for this problem will be proposed in Section 4.2.2.

Algorithm 1: Particle pose and initial covariance matrix prediction

$P_m = 0$
for *each particle m* **do**
  $s_t^m = f(s_{t-1}^m, u_t)$;
  $G_u = \frac{\partial f}{\partial(s_x, s_y, s_\theta)}$;
  $G_p = \frac{\partial f}{\partial(\delta_s, \delta_\theta)}$;
  $P_m = G_p P_m G_p^T + G_u Q G_u^T$;
end

2.2.2 Sampling a pose: particle update

Particle diversity is an important factor that determines the estimation accuracy. The algorithm uses a technique to deal with sample impoverishment. A new proposal distribution (set of particles generated after integrating the control data) is computed using the most recent measurement [10]. Then, a new particle pose is sampled; this is illustrated in Algorithm 2. The proposal distribution construction procedure can be parallelized on GPU pipelines, since each operation can be done independently for each particle.

Algorithm 2: Particle update

for *each particle m* **do**
  $\mu_0^m = s_t^m$, $\Sigma_0^m = P_m$;
  for $n \leftarrow 1$ **to** N **do**
    $\Sigma_n^m = \left[H_p^T Z_n^{-1} H_p + \left(\Sigma_{n-1}^m\right)^{-1}\right]^{-1}$;
    $\mu_n^m = \mu_{n-1}^m + \Sigma_n^m H_p^T Z_n^{-1}(z_t - \hat{z}_t)$;
  end
  $s_t^m \sim \mathcal{N}\left(\mu_n^m, \Sigma_n^m\right)$;
end

2.2.3 Estimation

During the estimation step, if a landmark is matched with a current observation, its position in the image $(\hat{u},\hat{v})$ is predicted using the pinhole model (3). Innovation between the predicted pose and the observation (u, v) is computed. The extended Kalman filter (EKF) corrects the inverse depth parameterization (ρ, θ, ϕ) of the matched landmark and its covariance matrix. Likelihood (the probability to observe the matched landmark from a given particle pose) is computed according to the observation. Since each particle has its own map, the operation mentioned above can be done independently and hence the estimation task can be parallelized on GPU.

$$h = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} c_u + f_{k_u} \frac{x_{a,\text{cam}}}{z_{a,\text{cam}}} \\ c_v + f_{k_v} \frac{y_{a,\text{cam}}}{z_{a,\text{cam}}} \end{pmatrix} \tag{3}$$

Algorithm 3: Monocular FastSLAM 2.0

while *1* **do**
  Prediction;
  for *each odometric acquisition* **do**
    $u_t \leftarrow (\delta_s, \delta_\theta)$;
    Predict particle pose and its covariance matrix. See Algorithm 1;
  end
  Image Processing;
  $(u_k, v_k) \leftarrow$ Fast Corner Detector;
  select high particle weight s_t;
  for ***Each*** *Landmark* **do**
    $\hat{z_n}\left(\hat{u_n}, \hat{v_n}\right) \leftarrow h\left(s_t, X_n\right)$;
    $zmssd\left(\left(\hat{u_n}, \hat{v_n}\right), (u_k, v_k)\right)$ Select landmark with small zssd;
  end
  Particle Update;
  $s_t^m \sim N(\mu_n^m, \Sigma_n^m)$ Sample new particle pose. See Algorithm 2;
  Estimation;
  for ***Each*** *Particle* **do**
    for ***Each*** *Landmark* **do**
      $(X, C)_n^m \leftarrow kalman\left(X_n^m, C_n^m, \hat{z_n}, z_n\right)$
      compute weight ω^m
    end
  end
  Initialization;
  for ***Each*** *Particle* **do**
    Compute $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$ Inverse depth parametrization;
  end
  Resampling;
  Importance resampling;
end

2.2.4 Initialization

During mapping, SLAM algorithms need to know initial landmark positions and the covariance matrix. This seems easy with SLAM algorithms based on LASER sensors since the observation model is invertible [13]. However, in the case of a monocular vision algorithm, the

landmark initialization is not obvious. A monocular camera is a projective sensor which cannot provide depth of a landmark in a scene. In order to estimate depth of a landmark, and so its position in the scene, the landmark must be tracked in more than one frame. In our implementation, we used the inverse depth initialization method [19].

This method allows adding a landmark from the first view. Initial coordinates of a landmark are $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$, where x_i and y_i are the first view camera poses, θ_i is the azimuth, φ_i is the elevation, and ρ_i is the inverse depth. This is depicted in Fig. 2.

2.2.5 Resampling

Resampling task deletes very improbable trajectories in order to maintain a constant number of particles and to prevent particle depletion. This depends on the weight of each particle computed in the estimation step [20]. In our implementation, the most time-consuming parts in the resampling step are parallelized on GPU pipelines. First, the total weight update task, in essence, starts before the resampling task and updates total particle weights based on likelihood computed in the estimation phase. The resulting weight is the product of likelihoods derived from each matched landmark $\left(w_t^M = w_t^M * \prod_{i=1}^{N_m} \omega_i\right)$. The weight normalization and weight summation $\left(w_{i_n} = \frac{1}{w_s}, w_s = \sum_{i=1}^{M} w_i\right)$ are also parallelized on GPU. Computation of the minimum number of effective particles before resampling can also be implemented in parallel $\left(N_{\text{eff}} = \frac{1}{\sum_{i=1}^{M} w_i^2}\right)$.

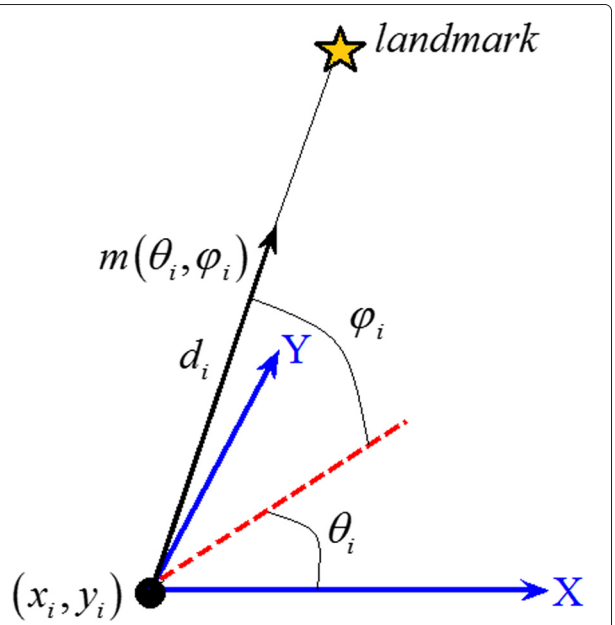

Fig. 2 Inverse depth parameterization of a landmark

3 Evaluation methodology

Our evaluation methodology consists on analyzing the algorithm and its dependencies. We identify the processing tasks that require considerable amount of time by evaluating their processing times. The algorithm is then partitioned into functional blocks (FBs) performing specific tasks. In order to have a bounded processing time, a threshold is fixed for each parameter. Functional blocks that impose the most important processing time are then optimized.

Such evaluation methodology is widely adopted when dealing with the implementation of compute-intensive algorithms such as SLAM. In [21, 22], Dine et al. adopted the same evaluation methodology to study the embeddability of the graph-based SLAM. In [23], the same methodology is used for an efficient implementation of the EKF-based SLAM on a low power-multiprocessor architecture.

3.1 Real dataset-based evaluation

SLAM algorithms are interesting applications when it comes to explore in a real indoor/outdoor environment. Evaluation of SLAM algorithms requires a set of different sensor data which are necessary for benchmarking. Sensor data can be obtained either by using a real instrumented robot or an available dataset. In our evaluation, we used a real indoor dataset [24]. This dataset provides a set of different sensor data. We have used data of encoders and a monocular camera.

3.2 Functional block partitioning

The algorithm is analyzed in terms of instruction and operation order which allows defining functional blocks (FB) described in Fig. 3. FB1 processes odometric data to calculate the future particle state and compute the related covariance matrix. Images are processed by FB2. This task extracts features from an image using FAST detector and performs a matching task between observations and landmarks. In our implementation, the matching task is performed according to the particle that has the high importance weight as proposed in [25]. Particles pose are then enhanced in FB3 based on the matched landmark computed in FB2. FB4 updates matched landmarks in FB2 using EKF. FB5 computes the initial inverse depth parameters for each new landmark. FB6 resamples new set of particles.

3.3 Algorithm dependencies and threshold definition

FB1 depends on the number of integrated odometric data at each iteration. FB2 depends on the number of landmarks in the particle map, the number of extracted features and the size of both images and corner descriptor. FB3 depends on the number of matched landmarks. FB4 depends on the number of matched landmarks to be

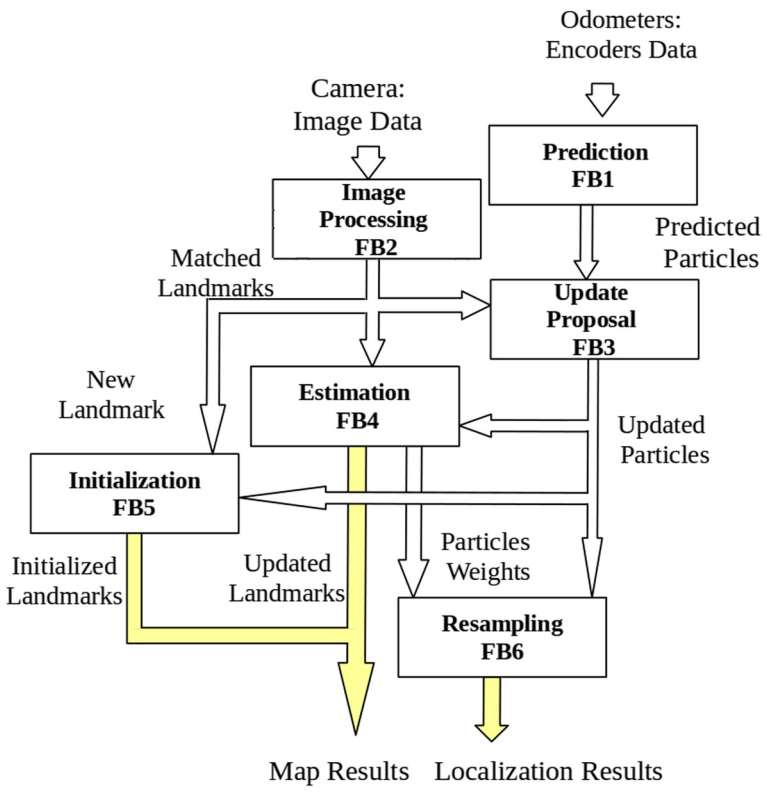

Fig. 3 Functional block partitioning

corrected. FB5 depends on the number of newly observed landmarks to be initialized. FB6 depends on the number of likelihoods computed in FB4 and the number of used particles.

In our experiments, we set a threshold value for each parameter in order to bound the processing time of each FB. We set the thresholds as follows:

- Number of odometric data is unbounded. It is related to encoder frequency used in experiment.
- Image size is fixed by dataset used in experiment: 320×240 pixels.
- Size of descriptor: 16×16 pixels.
- Maximum number of landmarks for each particle is set to 500. Note that this value is decreased for even larger number of particles to avoid exhausting memory.
- Maximum number of extracted features: 60.
- Maximum number of matched landmarks: 40.
- Maximum number of landmarks being initialized: 40.

3.4 Running times

Processing times of FBs depend on many parameters. Their dependencies determine the number of occurrences of a functional block per one iteration of the "while" loop (Algorithm 3). Processing times reported in this paper were computed relatively to the Mean of Occurrences per Time Stamps (MOTS). If a functional block FB_x is executed N_{FB_x} times in a single iteration, the MOTS is given by Eq. 4 where n is the number of iterations. Therefore, the mean of processing time t'_{FB_x} of a functional block FB_x is given by Eq. 5. t_{FB_x} is the execution time (an average of 10 runs of the algorithm for 500 iterations) of a single execution of FB_x.

$$\text{MOTS} = \frac{1}{n}\left(\sum_{i=1}^{n} N^i_{FB_x}\right) \tag{4}$$

$$t'_{FB_x} = \text{MOTS} * t_{FB_x} \tag{5}$$

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

4 Hardware software matching

Signal processing community has always been interested in implementing algorithms they developed. Evolution of the heterogeneous architectures and tools has allowed

designing complex systems that we have not even dare to consider few years ago. We have gradually moved from a separate study of algorithms and architectures, to a more formalized approach taking into account simultaneously algorithm and system architecture for an efficient matching. Algorithm architecture matching consists of a simultaneous study of the algorithm and the architecture in order to perform an optimized implementation of the algorithm taking into account different constraints (real time and embeddability).

Throughout this work, we aim to implement the FastSLAM2.0 algorithm on a heterogeneous embedded architecture. This requires a process to map the algorithm on the target architecture. A homogeneous implementation of FastSLAM2.0 algorithm was previously performed on a low power embedded architecture [26]. Nevertheless, such architecture do not have a suitable GPU to use for general purpose computing. New solutions have appeared using GPUs for general purpose computing. This is proved in the modern systems implementing heterogeneous architectures (HSA) allowing the use of GPUs and CPUs together. Sequential tasks run on CPU while the computational-intensive tasks are handled by GPU. In this article, an heterogeneous implementation of FastSLAM2.0 algorithm is performed using a modern embedded system on chip: the NVIDIA Jetson Tegra K1.

4.1 Hardware description

Tegra *K*1 is a recent system on a chip (*SoC*) developed by NVIDIA for mobile devices and multimedia applications. Figure 4 shows the block diagram of this system. K1 processor integrates a quad-core ARM Cortex A15 CPU and an NVIDIA Kepler GPU with 192 NVIDIA CUDA cores. Specifications of this architecture are gathered in Table 1.

4.1.1 Embedded GPGPU programming

Embedded GPU resources can be accessed by a programmer using various programming languages. CUDA presents the user with a C language for direct application development on NVIDIA GPUs which restrict code portability specifically to NVIDIA hardwares.

Tegra K1 is selected to evaluate our implementation. OpenCL on such architecture is not available. Moreover, it is important to note here that at the time of writing this paper, OpenCL drivers for other boards with embedded GPUs are not available in the public realm. Embedded boards that have the software support to use OpenCL for GPGPU programming contain GPUs with only low numbers of cores which are not really dedicated to high-performance computing. In [27], authors used OpenCL on a Vivante GC2000 GPU with 4 SIMD cores on the i.MX6 Sabre Lite development board. In [28], Maghazeh et al. used OpenCL on a Mali-T628 MP6 with 8 cores on the ODROID board and on a MALI-T604-MP4 GPU with 4 cores on the ARNDAL board. Therefore, using OpenCL in our work is not a good choice. Such boards with powerless embedded GPGPU may not be a good choice for a computationally intensive algorithm. As and when such boards, with a powerful embedded GPU and their respective software drivers supporting OpenCL, become available, it will be worthwhile to consider OpenCL as a programming language.

This work adopts OpenGL for developments. The majority of current embedded GPUs support OpenGL which is independent to a specific architecture. Adopting OpenGL for general purpose computing requires a prior experience in graphic programming. Although, it provides code portability between different embedded GPUs which makes the resulting implementation independent and allows conducting a comparative study between different embedded GPU architectures. Recent works adopted OpenGL for GPGPU programing. Hendeby et al. [6] used OpenGL for general purpose GPU particle filter. Weinlich et al. [29] and Oliveira et al. [30] presented a comparison between different GPGPU programming languages: OpenCL, OpenGL, and NVIDIA CUDA. They proved that by adopting the same optimization strategy among these languages, the gap is slightly different among them in terms of acceleration.

To use OpenGL for GPGPU, a typical GPGPU application program is based on three phases: (i) upload a suitable shader; (ii) allocate appropriate processing units for vertex and fragment shader; (iii) draw a suitable quad to trigger computation and download results.

4.1.2 Unified-shading architecture

Tegra K1 contains a recent GPGPU that adopts a unified shader architecture (Fig. 4). Unified shading architecture hardware is composed of an array of computing units (192 CUDA cores) which are capable of handling any type of shading tasks instead of dedicated vertex and fragment processor as in old GPUs. All computing units have the same characteristics. They can run either a fragment shader or a vertex shader. With a heavy vertex workload, we could allocate most computing units to run a vertex shader. In the case of a low vertex workload and a heavy fragment load, more computing units could be allocated to run fragment shader. In our work, we allocate more processing units to run the fragment shader to perform the desired parallel processing.

4.2 CPU-GPGPU partitioning

The HSA studied in our work allows the use of GPU and CPU together to enhance the global processing time. Using FB partitioning, we propose a distributed implementation of the monocular FastSLAM2.0. Functional blocks that require significant processing time are parallelized on GPU while sequential blocks are implemented

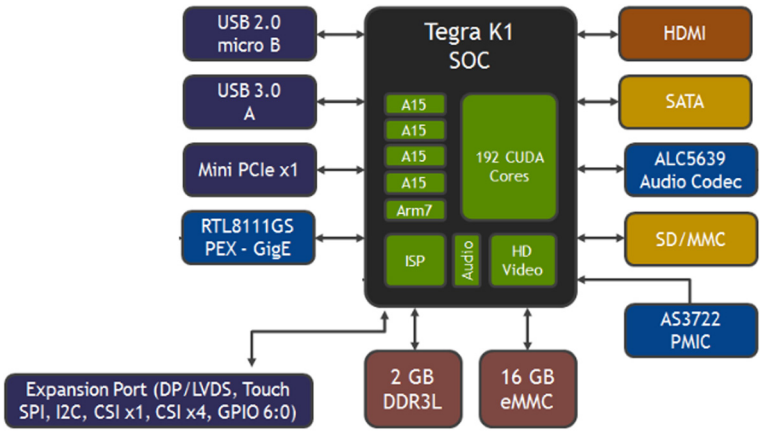

Fig. 4 Tegra K1 block diagram [33]

on CPU. Algorithm 4 describes the CPU-GPGPU partitioning of FastSLAM2.0. The implementation of each FB is described in the following sections.

Algorithm 4: CPU-GPGPU partitioning

while *1* **do**

$u_t \leftarrow$ Odometric data;

$z_t \leftarrow$ Camera data;

CPU	**GPGPU**
Time stamping	
Image processing	
	Prediction
	Particle update
	Estimation
	Resampling
	Initialization
Updating particles map	

end

4.2.1 CPU implementation of image processing task (FB2)

In image processing task, landmark detection is done using the FAST detector. We used an instance of the algorithm that is already optimized using machine learning [9]. Matching task is performed once using the highest weighted particle [25]. However, it can be parallelized according to the number of detected observations. In essence, at each time stamp, only few observations are detected. Therefore, the matching task is well implemented on ARM quad-core (more processing units are not needed). Implementing this task on GPU will only reduce performances. Transfer time will be larger than the execution time. Furthermore, particles map cannot be all transferred to GPU memory.

Table 1 NVIDIA Tegra K1 specifications

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

4.2.2 GPGPU implementation of prediction task (FB1)

Particle pose $s_t^m = (s_x, s_y, s_\phi)$ is transferred to a texture in global GPU memory. Each texel in texture memory holds on one particle state. In our implementation, we preferred generating random numbers on CPU for each particle and transfer them to a separate texture on GPU memory. This allows generating an accurate particle pose to surpass the problem discussed in Section 2.2.1. We could employ some techniques to achieve a parallel random number generation such as those described in [15], but it would be at the expense of localization accuracy. Poor particle distribution greatly affects localization results specially in large-scale environment.

Many encoder data are acquired at each time stamp (one iteration of the while loop in Algorithm 3). Therefore, the particle poses must be updated at each received encoder data to reconstruct properly the trajectory. To implement this process, a multiple rendering pass is needed. Texture is used as render target to store the output results for one prediction operation, and then it is directly used as input texture for the next operation. Since textures are either read-only or write-only, three textures are needed for FB1. One unchanged read-only texture is used for encoder data u. Two other textures are attached to frame buffer object (FBO): read-only texture s^t_{old} for input particle pose and write-only texture s^t_{new} to store predicted pose. The role of s^t_{new} and s^t_{old} is swapped since the value in s^t_{old} is no longer needed once new values have been computed.

This is illustrated in Algorithm 5. s is a table that holds two ping-pong texture identifiers ($s_{\text{old}}, s_{\text{new}}$). Role of these textures is changed within the loop by *swap*() function. *glDrawBuffer* sets writable textures. Two routines *glActiveTexture* and *glBindTexture* set readable textures. N is the number of odometric data, *drawQuad*() is a function that launches computing by drawing a suitable quad. The initial covariance matrix P_m is computed in the same way using multiple rendering pass.

Algorithm 5: Prediction using multiple rendering pass

```
attachement[0] = GL_COLOR_ATTACHMENT0;
attachement[1] = GL_COLOR_ATTACHMENT1;
read = 1, write = 0;
Attach s[0] to attachement[0], and s[1] to
attachement[1];
Transfer particles poses to s[1];
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE, u);
for m ← 1 to N do
    Transfer noisy encoders data to texture u;
    glDrawBuffer(attachement[ write] );
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_RECTANGLE,
    s[read]);
    drawQuad();
    swap(read, write);
end
```

4.2.3 GPGPU implementation of particle update task (FB3)

Algorithm 2 is parallelized on GPU using multiple rendering pass. Each render pass computes the mean distribution $\mu_t^m = (x, y, \theta)$ and its 3×3 covariance matrix Σ_t^m in parallel for each particle. A simple output texture will not be enough to store the computed values. A solution is to write to several output textures in one render pass. Four output textures are needed to store the mean and the covariance matrix computed for each particle. Therefore, twelve textures are used for a Gaussian construction distributed as follows:

- four textures, read-only not changed, are used for matched landmark parameters $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$.
- eight textures attached to the color attachment of FBO where
  - four of them are read-only used to hold the initial Gaussian $\left(\mu_{\text{old}}^{m,t-1}, \Sigma_{\text{old}}^{m,t-1}\right)$,
  - four write-only textures contain result of one render pass of the updated proposal distribution $\left(\mu_{\text{new}}^{m,t}, \Sigma_{\text{new}}^{m,t}\right)$.

This is depicted in Algorithm 6. First, eight ping-pong textures `pingpongTexID` are attached to FBO, then we set the four readable textures held in `LdmkTexID` array which contains single landmark parameter. The first loop transfers $\left(\mu_{\text{old}}^{m,t-1}, \Sigma_{\text{old}}^{m,t-1}\right)$ to four textures already attached to FBO. The second loop is the main loop that computes the new proposal distribution using multiple render pass for N-matched landmarks. We transfer at each iteration of the "For" loop the matched landmark parameters to textures in `LdmkTexID` array. `glDrawBuffers` routine sets writtable textures where the new computed Gaussian $\left(\mu_{\text{new}}^{m,t}, \Sigma_{\text{new}}^{m,t}\right)$ will be stored. The following loop sets readable textures where the old Gaussian mean and covariance will be read. For the next render pass, textures where the new computed Gaussian was written will be read-only, whereas textures that held the old Gaussian $\left(\mu_{\text{old}}^{m,t-1}, \Sigma_{\text{old}}^{m,t-1}\right)$ will be write-only. This is done by `swap()` function.

Algorithm 6: GLSL Gaussian construction

```
Attach eight ping-pong textures
pingpongTexID[0..7] to FBO;
Set the four readable textures LdmkTexID[0..3];
for i ← 0 to 3 do
    Transfer the initial particles Gaussian state
    μ_old^{m,t-1}, Σ_old^{m,t-1} to textures pingpongTexID[i];
end
for n ← 1 to N do
    Transfer the n landmark state to textures
    LdmkTexID;
    glDrawBuffers(4,pingpongTexID);
    for i ← 0 to 3 do
        sets readable textures in pingpongTexID;
    end
    drawQuad();
    swap();
end
```

It is important to note that we could allocate enough textures to hold all matched landmark parameters since we are allowed to input up to 32 textures. This would allow Gaussian construction to be done in one render pass. This reduces data transfer at each render pass which allows a significant improvement. However, unnecessary memory allocation should be avoided. Such implementation would increase memory requirements. So, the available on-chip memory will be almost entirely accessed by these textures.

4.2.4 GPGPU implementation of Estimation task (FB4)

Each render pass corrects one matched landmark in the map for each particle in parallel. Twelve textures are needed for FB4. Eight read-only textures are used as

inputs to the fragment shader to hold the old matched landmark state $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$ and the current state of particles $\left(s_t^m, P_t^m\right)$. Four write-only textures are attached to FBO to store the updated landmark state for each input particle.

The fragment shader implements general extended Kalman equations to update the landmark state and compute the corresponding likelihood. This is described in Algorithm 7. Each loop iteration corrects one matched landmark in the map for each particle in parallel. If more than one landmark are matched, many iterations are needed to update them. Results (updated landmark state) are transferred from textures to CPU.

Algorithm 7: GLSL estimation

for $n = 1$ **to** N **do**

Compute in parallel for each particle the updated state of the n **landmarks**:

- Transfer the matched landmark state $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$ to four read-only input textures;
- Draw a filled rectangle to trigger the computation
- Download the corrected landmark state from four write-only output textures to CPU

end

4.2.5 GPGPU implementation of inverse depth initialization task (FB5)

Five initial landmark parameters are calculated for all particles $(x_i, y_i, \theta_i, \varphi_i, \rho_i)$. The inverse depth parameter ρ has a constant value $\rho = 0.25$ [19]; hence, it is initialized on CPU. However, $(x_i, y_i, \theta_i, \varphi_i)$ are computed on GPU. Landmark position in current frame and particle poses are transferred to GPU via textures. Only two textures are needed for FB5: one read-only texture with RGBA internal format that holds the particle pose (one particle per texel $R = x_p, G = y_p, B = \theta_p, A = 0$) and one write-only texture attached to the frame buffer object to store initialized landmark parameters (one landmark per texel: $R = x_i, G = y_i, B = \theta_i, A = \varphi_i$). Landmark poses in image are loaded to the shader as a uniform value. To exemplify GLSL source code, Algorithm 8 describes fragment shader code needed for inverse depth initialization.

sampler2DRect is a specific pointer to the active texture unit. Hence, *partPose* points out input particle poses. The first line of the algorithm makes a texture look-up and retrieves the particle pose stored in a four-dimensional vector *pos_particle*. The next two lines compute consecutively the camera pose and the landmark position in the environment. The first view camera pose (x_i, y_i), the azimuth θ_i, and the elevation ϕ_i are stored in a specific variable referred to as `gl_FragColor`.

Algorithm 8: Fragment shader inverse depth initialization

```
uniform sampler2DRect partPose;
uniform vec3 trans, uniform vec2 uv, uniform mat3
rot vec3 pos_cam, pos_lmk, vec4 pos_particle;
void main(void) {
pos_particle = texture2DRect(partPose,
gl_TexCoord[0].st);
pos_came = rot*trans;
pos_lmk = compute_world_Ldmk_pose(uv);
gl_FragColor.x = pos_cam.x + pos_particle.x;
gl_FragColor.y = pos_cam.y + pos_particle.y;
gl_FragColor.z= - atan((pos_cam.x -
pos_lmk.x)/(pos_cam.z - pos_lmk.z));
gl_FragColor.w = atan(pos_lmk.y/sqrt
(pow(pos_lmk.x -pos_cam.x,2) + pow(pos_lmk.z -
pos_cam.z,2)));
}
```

4.2.6 GPGPU implementation of resampling task (FB6)

Implementation of FB6 is similar to FB1. One texture read-only contains the input likelihood score and two textures contain w_{old} and w_{new} attached to frame buffer (in this case, $w_{\text{new}} = score * w_{\text{old}}$). We switch roles of textures from read-only to write-only textures. The normalized weight is computed in one render pass; input texture that stores the updated weight is loaded to the shader and the sum of weights is loaded as a uniform value. Weight summation is a reduction-type operation that can be also implemented in parallel on GPGPU by mapping a $M \times M$ texture to 1×1 texture. The algorithm reduces recursively the output region size by computing the local summation of each 2×2 group of elements in one shader and writing it to the corresponding output location.

Figure 5 summarizes the first reduction step for a 4×4 input texture. The left texture shows the input texture. The gray area is the range of the output texels (which will be located in another texture). The right texture shows result of the first reduction pass. Each output texel contains the local summation of a corresponding 2×2 region in the input texture. Computation of the minimum number of effective particles before resampling is also implemented in the same way. To implement this on GPGPU, reduction is performed in a ping-pong manner. Three different textures are required: one input texture containing input weights w and two temporary textures to perform the reduction itself.

4.3 Partitioning model

The proposed distributed implementation on TK1 is described in Fig. 6. The overall block diagram of the

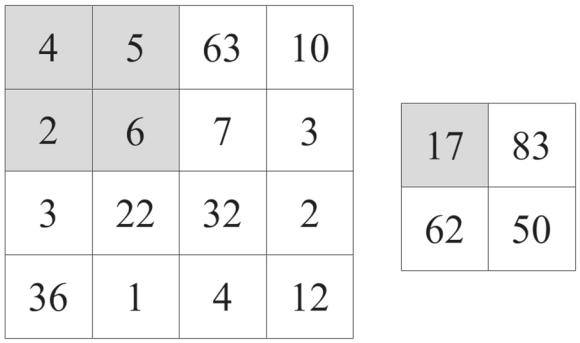

Fig. 5 Implementation of the weight summation on GPU. This is recursively repeated until a $M \times M$ texture is reduced to 1×1 scalar texture

massively parallel architecture is detailed in Fig. 7. It consists of a set of processing elements (PEs) operating in parallel and communicating with GPU memory. Figure 8 illustrates the case of four independent PEs corresponding to FB1, operating on odometric data. Figure 9 shows the internal block diagram of the *i*th processing element corresponding to FB4. This block describes the ordinary EKF update of the *j*th matched landmark relatives to the *i*th particle. The bottleneck in such heterogeneous implementation is the transfer time between CPU-GPGPU and memory access in the GPU side. This greatly reduces performances that can be obtained from HSA systems. Our parallel model has bandwidth advantage by managing data transfer and memory access in an efficient way.

4.3.1 Data transfer management

Data transfer is managed by PBO (pixel buffer object) through asynchronous DMA transfer. CPU involves only loading data to PBO but not data transfer from PBO to textures. Instead, the memory controller manages data transfer from PBO to textures performing a DMA transfer operation without wasting CPU cycles. Without DMA, CPU is typically fully occupied during the transfer operation since data transfer comes before and after each GPGPU kernel execution. Thus, the CPU is unavailable to perform other tasks. The CPU needs to perform a time stamping task and processes images at the frame rate while arranging and updating the particle map. Using DMA, the CPU initiates first the transfer, then it performs time stamping and image processing tasks while the transfer is in progress. This is an advantage allowing efficient processing between CPU and GPU together. Figure 10 shows bandwidth improvement of PBO over generic data transfers on TK1. PBO-accelerated transfer is faster than the conventional transfer (1.2 GB/s against 640 MB/s).

4.3.2 Optimizing memory access

GPU memory access is optimized using texture memory which fully leverage the parallel processing power. In our implementation and during kernel computation, texture memory access patterns has a spatial locality. In other words, a processing unit running a fragment shader is likely to read from an address near the address that nearby processing unit read. Texture memory is cached on a chip and has a specialized caching scheme optimized for spatial locality which provides effective bandwidth advantage and reduce memory access.

5 Experimental results

In this section, we analyze the processing times of the proposed heterogeneous implementation discussed in Section 4. The experimental tests as well as the evaluation of processing times were based on our methodology discussed in Section 3. First, all processing times reported in our work as well as the localization results are obtained using a real dataset (Section 3.1). Second, we separately evaluate each functional block to synthesize the full FastSLAM2.0 implementation results (Section 3.2). Third, since each FB has parameter dependencies, evaluation was conducted based on a set of defined thresholds

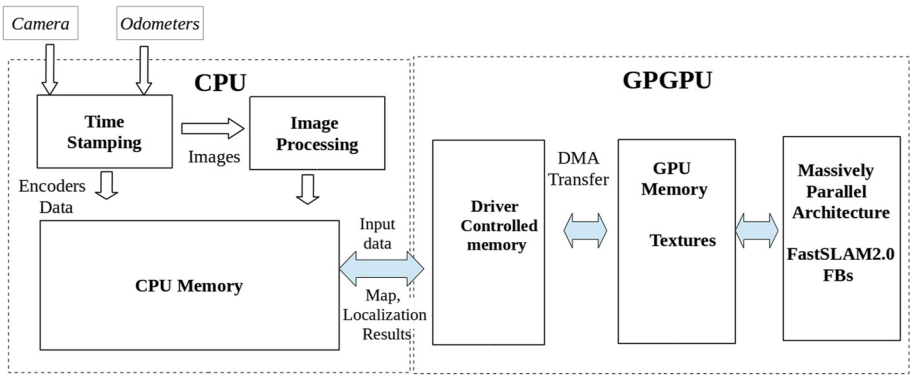

Fig. 6 Heterogeneous distribution of FastSLAM2.0

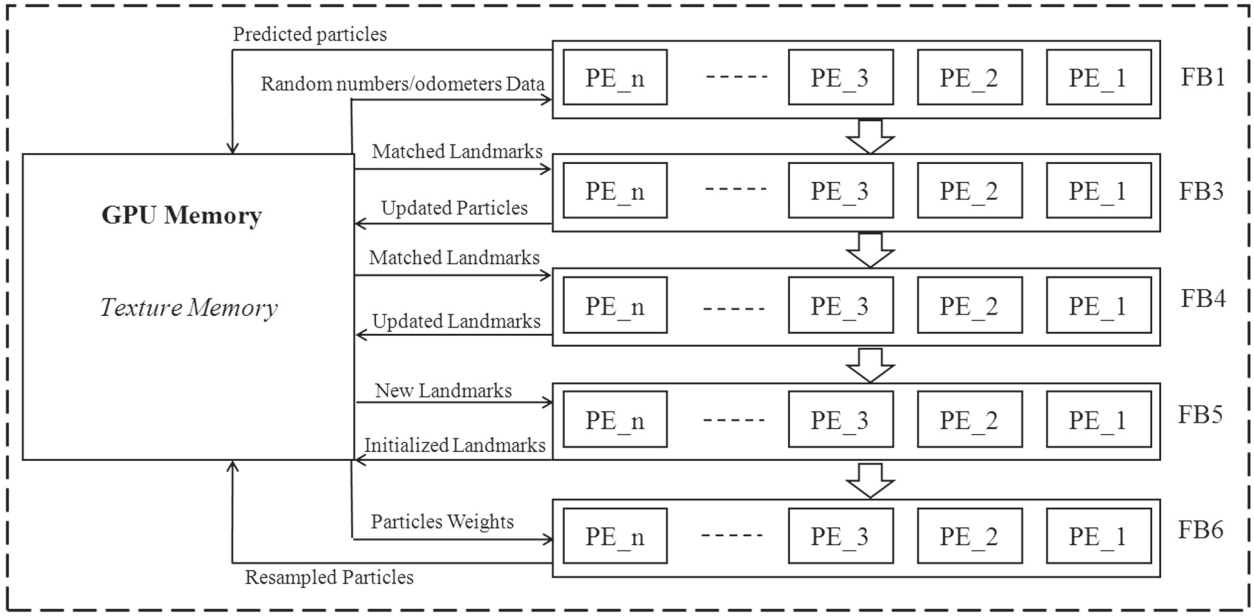

Fig. 7 Internal block diagram of the massively parallel architecture

to bound the computation time (Section 3.3). Finally, a FB may occur once or more time in a single iteration depending on its parameters. We report the mean of processing time t'_{FB_x} and t_{FB_x} respectively, computed relatively for the MOTS and for a single occurrence (Section 3.4).

5.1 Algorithm evaluation

The aim of this section is to evaluate FastSLAM2.0 on an embedded architecture and to determine the number of particles needed for a monocular FastSLAM2.0 to operate well in a large-scale environment (Fig. 11). Results are shown in Fig. 12. Since odometry data are very noisy, the odometric trajectory reconstructed by the prediction step (red line) diverges. FastSLAM2.0 (green line) provided a trajectory which follows the ground truth (blue line) along the explored area. A high number of particles in a monocular FastSLAM2.0 system is necessary to maintain a reasonable estimate of pose and landmark uncertainties. This greatly decreases the error of localization specially when exploring a large area (more details are given in Section 5.2). To evaluate the quality of localization, we measure the euclidean error between the real

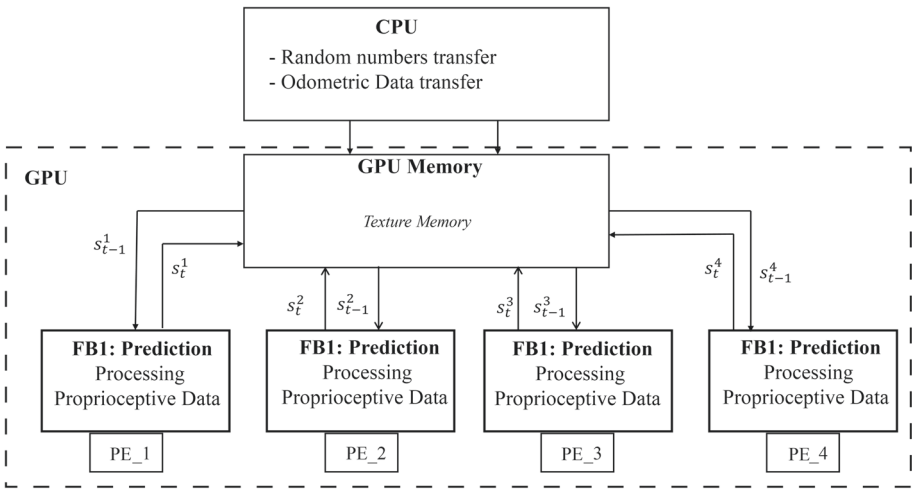

Fig. 8 Processing element distribution of FB1

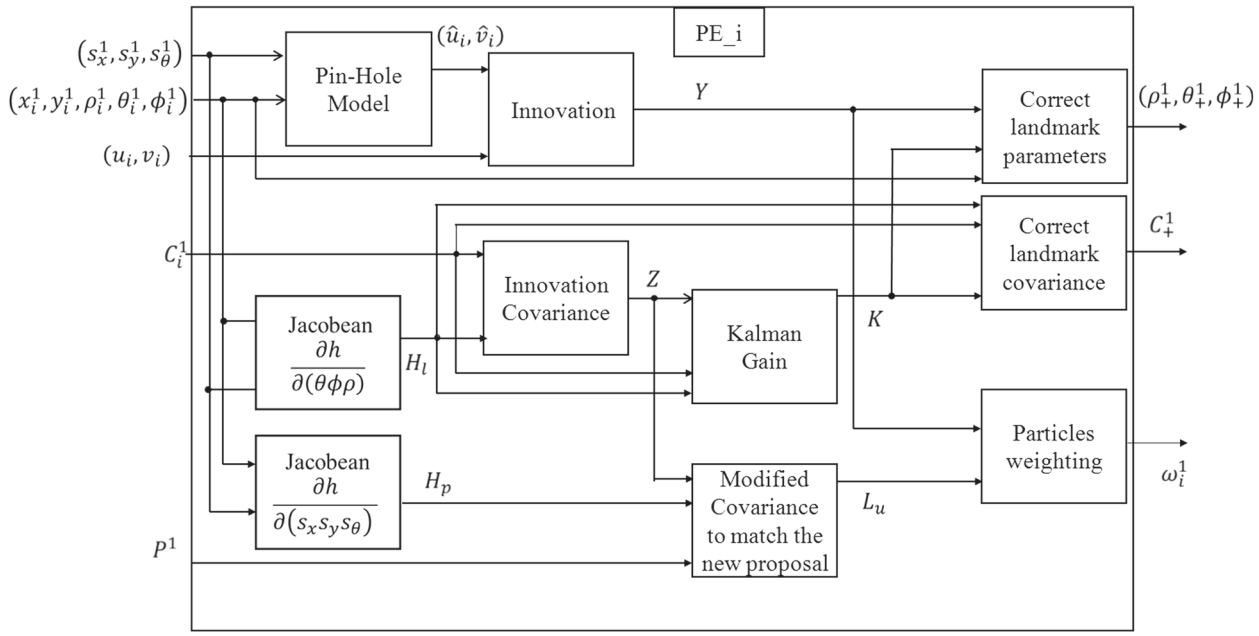

Fig. 9 The *i*th processing element of FB4

robot pose and the estimated pose (6). Figure 13 shows the effect of the particle number on the localization error. The error decreases as the number of particles increases.

$$d = \sqrt{(x_{\mathrm{GT}} - s_x)^2 + (y_{\mathrm{GT}} - s_y)^2} \tag{6}$$

(s_x, s_y) is the estimated robot pose and $(x_{\mathrm{GT}}, y_{\mathrm{GT}})$ is the reference pose.

5.2 Particle-wise GPU parallelization

Montemerlo et al. [10] implemented the FastSLAM2.0-based laser range finder that provides range and bearing of a landmark in a scene. Such algorithm can converge with few particles. The SLAM algorithm implemented in our work uses a monocular camera (bearing-only sensor) to observe the environment. The number of particles in such system is necessary to maintain reasonable estimates of pose and landmark uncertainty as stated in [8]. In

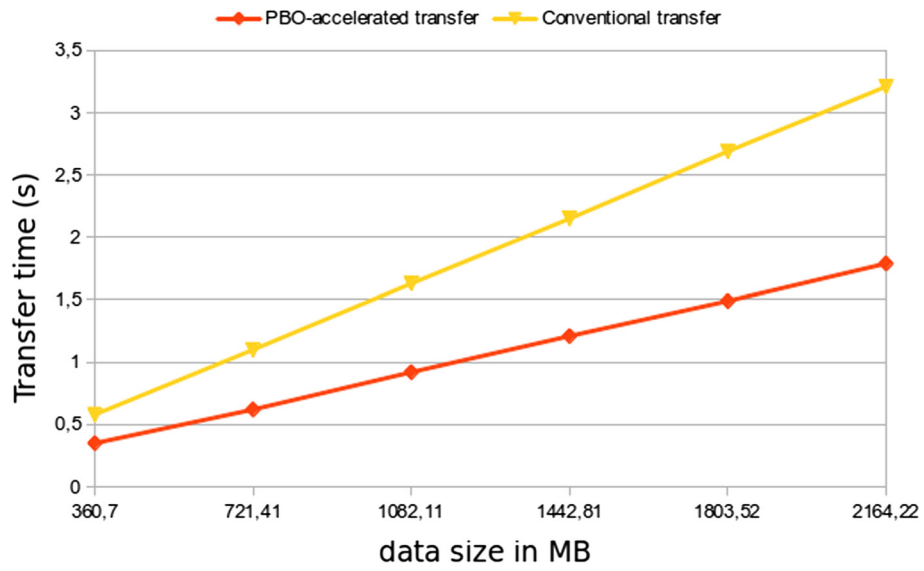

Fig. 10 CPU-GPU data transfer time

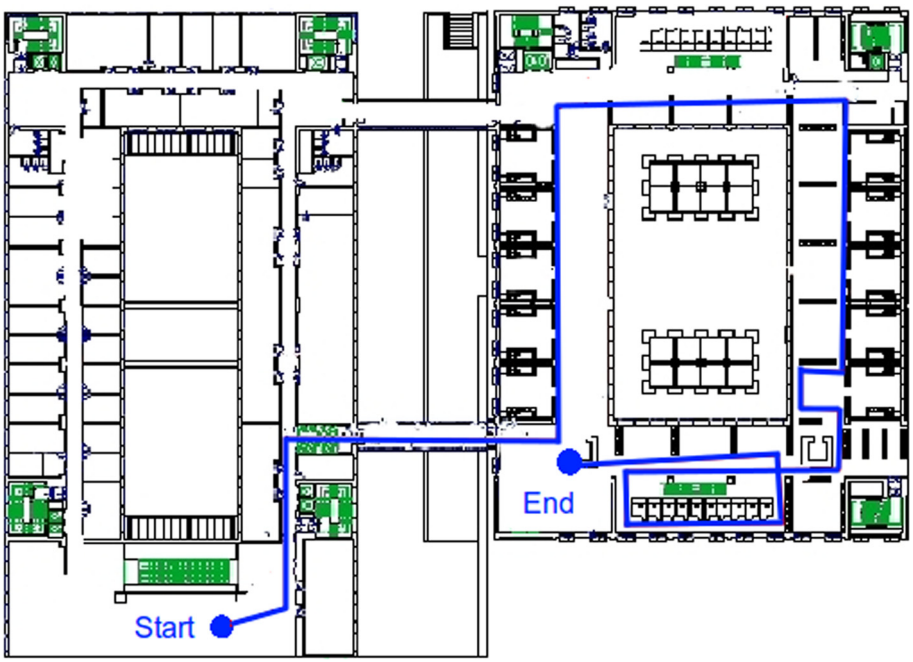

Fig. 11 Trajectory of the explored indoor environment

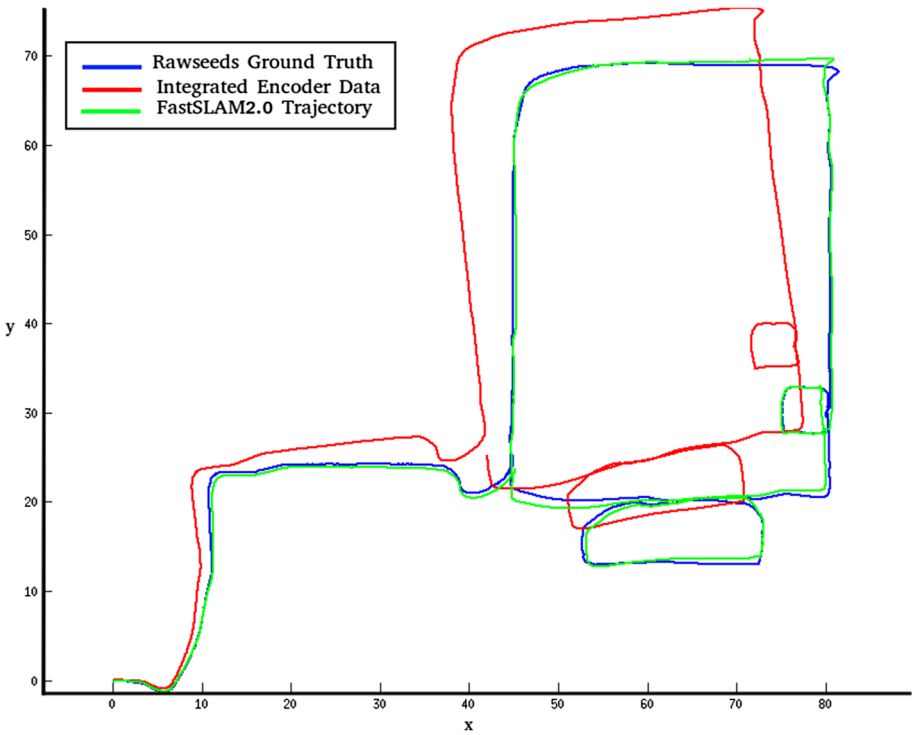

Fig. 12 FastSLAM2.0 results on Rawseeds dataset

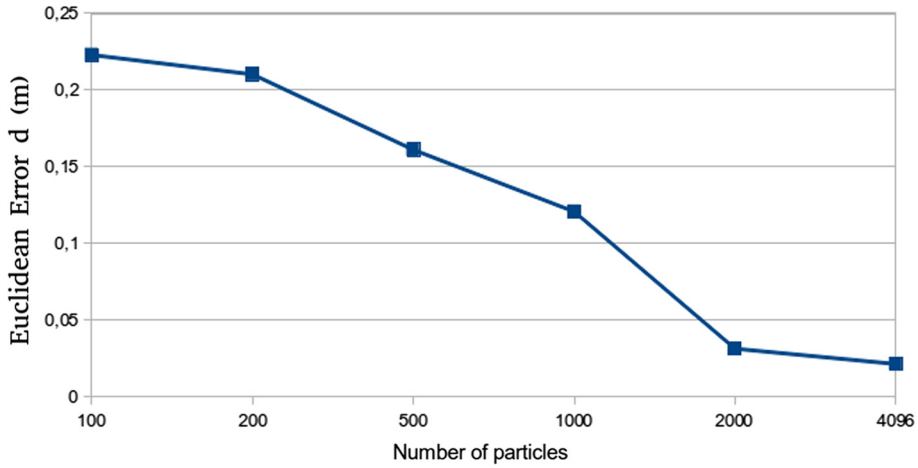

Fig. 13 Evolution of the localization error

addition, Eade and Drummond [8] conduct experiments with 50, 250, and 1000 particles to evaluate the impact of the number of particles on landmarks and pose estimation. They showed that 50 particles are sufficient for a very short sequence with few images. However, more detailed and rigorous analysis will be necessary for long trajectories and large environments. There remains significant challenges to tackle with FastSLAM2.0-based bearing-only sensor intended to operate in large geographic scales. The exact number of particles necessary is not yet defined which may increase with environment complexity.

In our evaluation (Section 5.1), we have used a very long indoor sequence with 5000 images. To close the loop over this larger trajectory, the number of particles must be increased for an accurate estimate of uncertainties. Our tests show that the monocular system can close the loop over this large trajectory (Fig. 12) using more than 500 particles. The more the number of particles increases, the more the landmark estimates are accurate and localization error decreases (Fig. 13).

5.3 Processing time evaluation

The one-core, quad-core, and CPU-GPGPU implementations were run and time evaluated for 500 iterations using Rawseeds dataset discussed in Section 5.1. Computing performances on one-core and quad-core CPU are used as a baseline to analyze the GPGPU acceleration results. As discussed in Section 5.2, a large number of particles is needed to achieve an accurate localization results. Therefore, we choose to conduct experiments with the number of particles that gives more accurate results.

Figure 14 presents the workload of each FB when running the one-core implementation with 4096 particles. FB1 is a time-consuming functional block. FB1 occurs several times in one iteration as many encoder data are acquired to reconstruct robot trajectory using motion model. Also, computing the initial covariance matrix P_m is time consuming. It increases as the number of odometric data increases between two consecutive measurements (image acquisitions). FB3 and FB4 occur only when there is at least one matched landmark. Therefore, their complexity depends on the number of matched landmarks. FB5 occurs only when there is a new landmark to add in the map. FB6 resamples particles only when there is at least one likelihood computed in FB4 to reflect observation on particle poses.

Table 2 synthesizes a comparison of processing times and global acceleration of each major kernel function obtained after parallelization when running the algorithm

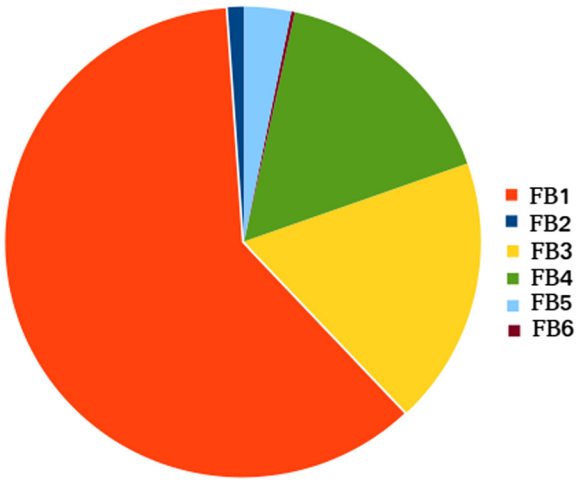

Fig. 14 FBs workload on Rawseeds dataset

Table 2 Mean of processing times of functional blocks on Rawseeds dataset

Functional blocks (FBs)	Mean of processing time per one occurrence t_{FB_x} (ms)				MOTS	Mean of processing times t'_{FB_x} (ms)				Speedup
	One-core	Quad-core	CPU-GPU			One-core	Quad-core	CPU-GPU		–
	–	–	Quad-CPU	GPU	–	–	–	Quad-CPU	GPU	
FB2	6.38	4.83	4.83	-	1	6.38	4.83	4.83	–	1.32
FB1	272.03	150.19	–	3.42	15.1	4107.8	2267.9	–	51.66	43.9
FB3	11.78	5.72	–	2.28	14.1	166.09	80.71	–	32.17	2.50
FB4	7.12	5.04	–	1.37	14.1	100.50	71.09	–	19.45	3.65
FB5	5.45	2.95	–	0.96	10	55.45	29.58	–	9.65	3.06
FB6	4.7	1.88	–	0.51	5.2	24.87	9.8	–	2.7	3.6
Total	307.46	170.61	13.37		–	4461.09	2463.91	120.46		20.38

with 4096 particles. For a mean of occurrences equal to 15.1, FB1 is executed in 4107.8 ms on one core. In quad-core implementation, FB1 is executed in 2267.9 ms. FB3 takes place only if there is at least one matched landmark. For a *MOTS* equal to 14.1, FB3 is executed in 166.9-ms on one-core CPU and 80.71 *ms* on quad-core CPU. After accelerating FB1 and FB3 on GPU, the average time of FB1 decreases from 2267.9 to 51.66 ms. This represents an acceleration of 44 times. For FB3, the average time dropped from 80.71 to 32.17 ms which results to 2.5 times improvement. FB4 is executed in 100.5 ms on one-core CPU, 71.09 ms on quad-core CPU, 19.45 ms on GPU for a *MOTS* equal to 14.1. FB5 takes 55.45 ms for a *MOTS* equal to 10 on one-core CPU, 29.58 ms on quad core, and 9.65 ms on GPU. Finally, particles are resampled in FB6 when a likelihood is computed. FB6 is executed in 24.87 ms on one-core CPU, 9.8 ms on quad-CPU, and 2.7 ms on GPU for a *MOTS* equal to 5.2. Processing time of CPU-GPU implementation has been decreased by a factor of 21.77 compared to OpenMP implementation on ARM quad core.

We note here that the prediction step (FB1) gets 44× speedup using GPGPU. This is because the parallel implementation of this block on GPU is done with much less data transfer between CPU and GPU. For M particles, we transfer only M random numbers in single-precision floating-point format from CPU to GPU and there is no transfer back to CPU from GPU. Other functional blocks (FB3, FB4, FB5, and FB6) require data transfer from CPU to GPU and back from GPU to CPU. In FB3, we transfer at each iteration the particle map (matched landmarks and their related covariance matrix) to GPU to update the particle pose. In FB4, we transfer at each iteration the matched landmarks and their covariance matrix to GPU to correct their states and we transfer back the updated landmarks to CPU. For FB5, we transfer unmatched landmarks to GPU. Once they are initialized, we transfer them back to CPU. In FB6, we transfer the computed likelihoods to GPU, then we transfer back to CPU the resampled particles. This reduces the gain that can be obtained after parallelization.

For further analysis of the algorithm dependencies, we run the one-core, quad-core, and CPU-GPGPU implementations for 500 iterations using a set of different particles ranging from 2^4 to 2^{16}. Note that a low number of particles (2^4) are not enough for accurate localization, not as much as (2^{14} or 2^{16}) are needed. Although, this range allows a better analysis of dependencies. Figure 15 shows the processing time against the number of particles in log scale coordinates to better show results.

As seen before, each FB of the algorithm depends on many parameters. The complexity of FB3 and FB4 increases as there are many matched landmarks to process. The decision to consider whether a landmark is matched or not is greatly related to uncertainty in robot pose [31]. As this uncertainty is varying with the number of particles used, due to the randomization applied to motion model, the number of matched landmarks is also varying for each implementation. In other words, the number of matched landmarks is not necessarily the same, neither when running the algorithm on one-core, quad-core, or GPU nor when running it for different numbers of particles. This is approved in Fig. 15. For the GPGPU implementation of FB3 and FB4, the number of matched landmarks with 2^{12} particles is higher than the number of matched landmarks with 2^{14}. Therefore, GPU can process FB3 and FB4 with 2^{14} particles even faster than it processes FB3 and FB4 with 2^{12} particles. This is because the complexity still rises since FB3 and FB4 are being executed sequentially within each computing kernel. Also, the one-core and quad-core CPU can process FB3 and FB4 slightly faster with 2^{12} than with 2^{10} particles for the same reason.

The complexity of FB5 increases as there are many new landmarks to initialize and to add in the map. The decision whether to add a new landmark or not is also related to robot pose uncertainty and to the number of particles used. Therefore, the number of new initialized landmarks

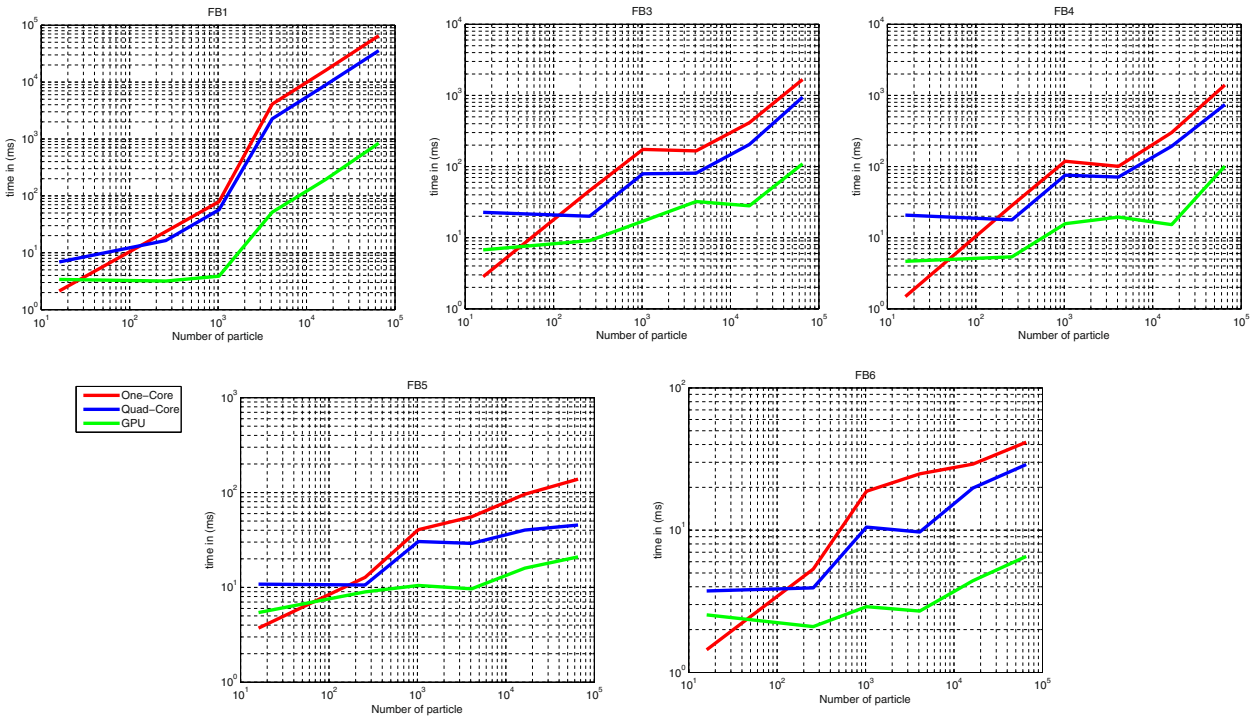

Fig. 15 FB dependencies on algorithm parameters

is also varying from an implementation to another one. This is seen in Fig. 15. GPU and quad-core CPU can process FB5 slightly faster with 2^{12} than with 2^{10} particles. The number of new landmarks decreases when implementing the algorithm with 2^{12} particles. When the number of new landmarks steadily increases, the processing time of FB5 scales linearly with the number of particles, as the case of the one-core implementation of FB5.

Since the number of matched landmarks is varying, the number of computed likelihood is also varying. Moreover, if this likelihood presents an outlier value, it is discarded and not used in FB6 to resample particles. This strict policy can decrease the number of the computed likelihoods and hence the processing time of FB6 also decreases. Figure 15 shows this dependencies. The processing time of FB6 GPU implementation decreases with 2^8 and 2^{12} particles. Also, for FB6 quad-core implementation, the processing time decreases with 2^{12} particles.

Unlike other functional blocks, FB1 depends only on the number of odometric data to process at each iteration. The number of odometric data provided by the sensor at each iteration is the same and remains independent of the implementation type that is currently running. Therefore, the complexity of FB1 scales linearly with the number of particles (Fig. 15).

Actually, such dependencies are inevitable, especially when dealing with a SLAM based on a probabilistic approach. A solution is to bound the processing time by defining a threshold value for each parameter.

To explore the parallel computing power of the embedded GPGPU, we report the overall processing time of the monocular FastSLAM2.0 for the three implementations. Different numbers of particles are used to better study the complexity. When using the same dataset and the number of particles, the speedup achieved by the GPU implementation compared to one-core and quad-core implementations, respectively, is shown in Fig. 16 (in log scale coordinate).

For few particles (2^4), CPU implementation performs better than quad-core and GPU implementations. The quad-core implementation seems not to perform well for few numbers of particles (2^4 and 2^8). This is due to the fact that many data are shared between threads and memory access is sequential among different threads, specially synchronization barriers among the currently running threads. This issue is common and well known when dealing with OpenMP implementation. This results to performance degradation. In this case, one-core implementation gets the best performance. Parallelizing with more threads on quad-core CPU only worths when the number of particles increases (2^{10} to 2^{16}). We can then achieve the best

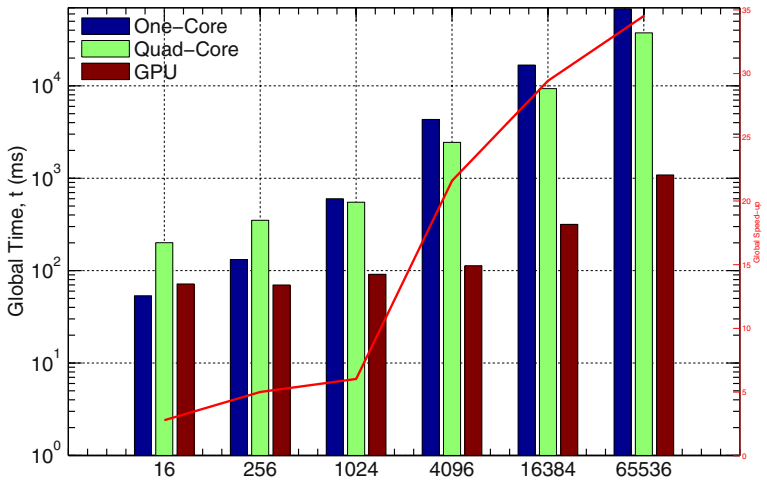

Fig. 16 Global processing time and overall speedup

performance according to one-core implementation. With 2^4 particles, GPU implementation also performs worse. This is due to data transfer time which is more important than the processing time. However, for even more particles, the degree of parallelization becomes important and hence GPGPU implementation is faster than one-core and quad-core implementations.

Figure 17 shows the evolution of CPP (cycle per particle) for different implementations computed with the following equation:

$$\mathrm{CPP} = \frac{f * t}{M}$$

M is the number of particles, f is the clock frequency expressed in *Hz*, and t is the processing time in *s*. CPP serves as a baseline for performance comparison of implementations on different processing units [32].

For one-core and quad-core implementations, the CPP value achieves the lowest point with 2^{10} particles. This is due to the fact that the global processing time t for both the one-core and quad-core implementations increases significantly with 2^{12} particles (see Fig. 16). CPU needs more cycles to process 2^{12} particles than it needs with 2^{10} particles. This is related to memory copy in the resampling step when a particle is duplicated. The memory copy is a time-consuming operation when there are many

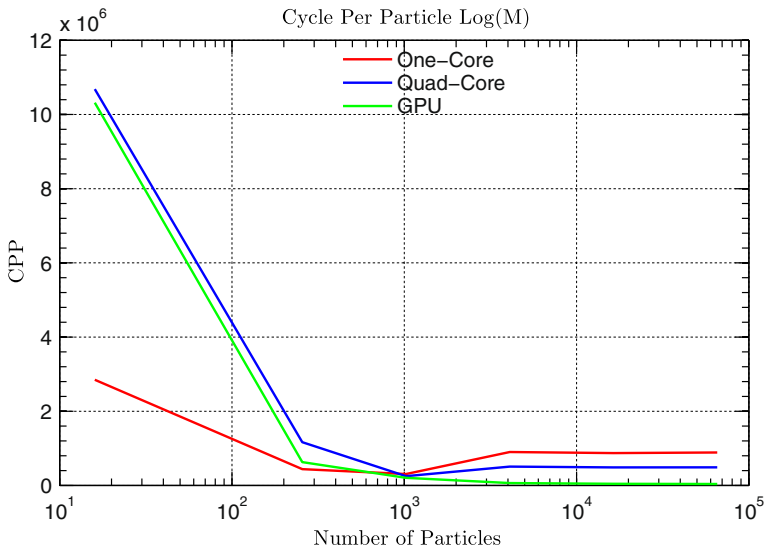

Fig. 17 Cycle per particle, Log(M)

landmarks in the map. This increases significantly the global processing time specially when many particles are used. This is the case when the global processing time increases significantly with 4096 particles. For even large number of particles (2^{14} and 2^{15}), we decrease the maximum number of landmarks in the map of each particle as mentioned in Section 3.3 to avoid exhausting memory usage. This reduced the memory copy operation in the resampling step and hence the global processing time steadily increases with the number of particles in the filter.

Contrary to GPGPU implementation, the global time slightly increases with 2^{12} particles. In general, CPP corresponding to GPGPU implementation keeps a lower value than one-core and multi-core implementations as long as the number of particles increases.

6 Conclusions

This article proposed an efficient matching of monocular FastSLAM2.0 algorithm on a heterogeneous embedded architecture. The first complete parallel FastSLAM2.0 implementation in literature on a embedded GPGPU is described. Using a real dataset, the parallel CPU-GPU implementation is shown to outperform a quad-core CPU implementation for many particles while maintaining the same localization accuracy.

The absolute performance of FastSLAM2.0 on an embedded GPGPU relies on the number of embedded cores (i.e., processing elements). As the number of processing elements steadily increases and can be expected to match the number of particles needed for an accurate monocular FastSLAM2.0 system intended to operate in large-scale environment, GPGPU is an interesting alternative architecture for monocular FastSLAM2.0 implementations. For a fixed number of particles and sufficiently large number of embedded cores, the parallel implementation will always be more efficient. Our FastSLAM2.0 GPGPU implementation has achieved up to 20× speedup with 192 GPGPU cores. It leads to a system that runs in real time and processes images at the frame rate they were acquired (30 FPS). This meet performance requirement of a robot to operate in real time.

It is important to note here that, at the time of writing this paper, Tegra K1 is the only embedded board that has a powerful GPGPU with 192 cores. Other boards with embedded GPUs only contain limited numbers of cores. As a conclusion, the monocular FastSLAM2.0 is shown to perform well with high number of particles in a large-scale environment. A naive implementation of FastSLAM2.0 with high number of particles would increase the localization accuracy but at the expense of robot performance to operate in real time. Our accelerated implementation on an embedded GPGPU achieved a compromise between accurate localization and real-time performance. Our results demonstrated that an optimized monocular FastSLAM2.0 partitioned on a heterogeneous embedded architecture is suitable to have high-speed execution and accurate results under real-time constraints in large-scale environments.

Appendix

Table 3 contains the parameter definitions.

Table 3 Parameter definitions

Parameters	Definition
zmssd	Zero mean sum of squared differences
m_d	Pixels mean of the landmark descriptor
I_{lmk}	Pixel intensity of the landmark descriptor
m_p	Pixels mean of the corner descriptor
I_p	Pixel intensity of corner descriptor
x, y	Location in image of the detected corner
i, j	Indexes to surrounding pixels in the descriptor
f	Motion model
h	Pin-Hol model
$u_t\ (n_l, n_r)$	Odometry data
$s_t\ (s_x, s_y, s_\theta)$	Particle pose
δ_s, δ_θ	Longitudinal and angular displacement
b	Wheels base
P_m	Particles initial covariance matrix
G_u	Jacobian matrix of motion model f derived according to s_t
G_p	Jacobian matrix of motion model f derived according to δ_s, δ_θ
Q	Motion model noise
M	Number of particles
N	Number of landmarks
μ	Mean of the new proposal distribution
Σ	Covariance matrix of the new proposal distribution
H_p	Jacobian matrix of observation model (Pin-Hol)
Z_n	Innovation covariance
z_t	Measurement
$\hat{z}_t$	Measurement prediction
$(\hat{u}, \hat{v})$	Predicted landmark position in image
(u, v)	Landmark position in image
$(x, y, \rho, \phi, \theta)$	Landmark inverse depth parametrization
(c_u, c_v, f_{k_u})	Standard camera calibration
$X\ (x_{\text{cam}}, y_{\text{cam}}, z_{\text{cam}})$	Landmark 3D world coordinate
C	Landmark covariance matrix
ω	Particle weight
N_{eff}	Number of effective particles

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Institut d'Electronique Fondamentale, Université Paris-Sud, 91405 Orsay, France. [2]Équipe Signaux-Systèmes et Informatique, ENSA, 1136 Agadir, Morocco. [3]École Nationale des Sciences Appliquées ENSA, 575 Marrakech, Morocco.

Received: 2 July 2015 Accepted: 29 July 2016
Published online: 17 August 2016

References

1. S-A Li, C-C Hsu, W-L Lin, J-P Wang, in *IEEE International Conference on System Science and Engineering (ICSSE)*. Hardware/software co-design of particle filter and its application in object tracking (IEEE, New York, 2011), pp. 87–91
2. M Moyers, D Stevens, V Chouliaras, D Mulvaney, in *IEEE International Conference on Eletronics Circuits and Systems (ICECS), Tunisia*. Implementation of a fixed-point FastSLAM 2.0 algorithm on a configurable and extensible VLIW processor (IEEE, Sfax, 2009)
3. TC Chau, X Niu, A Eele, W Luk, PY Cheung, J Maciejowski, in *Reconfigurable Computing: Architectures, Tools and Applications*. Heterogeneous reconfigurable system for adaptive particle filters in real-time applications (Springer, Los Angeles, 2013), pp. 1–12
4. O Tosun, *et al*, in *Intelligent Vehicles Symposium (IV), 2011 IEEE*. Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures (IEEE, Baden-Baden, 2011), pp. 820–826
5. S Maskell, B Alun-Jones, M Macleod, in *IEEE Nonlinear Statistical Signal Processing Workshop*. A single instruction multiple data particle filter (IEEE, Cambridge, 2006), pp. 51–54. doi:10.1109/NSSPW.2006.4378818
6. G Hendeby, R Karlsson, F Gustafsson, Particle filtering: the need for speed. EURASIP J. Adv. Signal Process. **2010**, 22–1229 (2010). doi:10.1155/2010/181403
7. H Zhang, F Martin, in *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*. CUDA accelerated robot localization and mapping (IEEE, Woburn, 2013), pp. 1–6
8. E Eade, T Drummond, in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Scalable monocular slam, vol. 1 (IEEE, New York, 2006), pp. 469–476. doi:10.1109/CVPR.2006.263
9. E Rosten, R Porter, T Drummond, Faster and better: A machine learning approach to corner detection. IEEE Trans. Pattern Anal. Mach. Intell. **32**(1), 105–119 (2010)
10. M Montemerlo, S Thrun, D Koller, B Wegbreit, in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*. FastSLAM2.0 an improved particle filtering algorithm for simultaneous localization and mapping that provably converges (IJCAI, Acapulco, 2003)
11. E Seignez, M Kieffer, A Lambert, E Walter, T Maurin, Real-time bounded-error state estimation for vehicle tracking. IEEE Int. J. Robot. Res. **28**, 34–48 (2009)
12. M Abouzahir, A Elouardi, S Bouaziz, R Latif, T Abdelouahed, in *2014 Second World Conference On Complex Systems (WCCS)*. An improved Rao-Blackwellized particle filter based-slam running on an OMAP embedded architecture (IEEE, Agadir, 2014), pp. 716–721. doi:10.1109/ICoCS.2014.7061001
13. S Thrun, Probabilistic robotics. Commun. ACM. **45**(3), 52–57 (2002)
14. P Hellekalek, Good random number generators are (not so) easy to find. Math. Comput. Simul. **46**(5), 485–505 (1998)
15. A De Matteis, S Pagnutti, Parallelization of random number generators and long-range correlations. Numerische Mathematik. **53**(5), 595–608 (1988)
16. CJK Tan, The PLFG parallel pseudo-random number generator. Futur. Gener. Comput. Syst. **18**(5), 693–698 (2002)
17. P Hellekalek, in *ACM SIGSIM Simulation Digest*. Don't trust parallel Monte Carlo!, vol. 28 (IEEE Computer Society, Banff, Alberta, 1998), pp. 82–89
18. M Sussman, W Crutchfield, M Papakipos, in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Pseudorandom number generation on the GPU (ACM, New York, 2006), pp. 87–94
19. J Civera, AJ Davison, JMM Montiel, Inverse depth parametrization for monocular SLAM. IEEE transactions on robotics. **24**(5), 932–945 (2008)
20. WG Madow, LH Madow, On the theory of systematic sampling, i. The Annals of Mathematical Statistics. **15**(1), 1–24 (1944). doi:10.1214/aoms/1177731312
21. A Dine, A Elouardi, B Vincke, S Bouaziz, in *2015 IEEE International Conference On Robotics and Automation (ICRA)*. Graph-based slam embedded implementation on low-cost architectures: a practical approach, (2015), pp. 4612–4619. doi:10.1109/ICRA.2015.7139838
22. A Dine, A Elouardi, B Vincke, S Bouaziz, in *2015 IEEE 26th International Conference On Application-specific Systems, Architectures and Processors (ASAP)*. Speeding up graph-based slam algorithm: a GPU-based heterogeneous architecture study (IEEE, Toronto, 2015), pp. 72–73
23. B Vincke, A Elouardi, A Lambert, Real time simultaneous localization and mapping: towards low-cost multiprocessor embedded systems. EURASIP J. Embedded Syst. **2012**(1), 1–14 (2012)
24. A Bonarini, W Burgard, JD Tardos, in *Proceedings of IROS'06 Workshop on Benchmarks in Robotics Research*. Rawseeds: Robotics advancement through web-publishing of sensorial and elaborated extensive data sets (Proceedings of IROS'06, Beijing, 2006)
25. E EADE, Monocular simultaneous localization and mapping. PhD thesis (2008)
26. M Abouzahir, A Elouardi, S Bouaziz, R LATIF, A Tajer, in *IEEE. The 13th International Conference on Control, Automation, Robotics and Vision, ICARCV*. FastSLAM2.0 running on a low-cost embedded architecture, (Marina bay Sands, Singapour, 2014)
27. A Maghazeh, UD Bordoloi, P Eles, Z Peng, in *2013 International Conference On Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*. General purpose computing on low-power embedded gpus: has it come of age? (IEEE, Agios Konstantinos, 2013), pp. 1–10
28. L Nardi, B Bodin, MZ Zia, J Mawer, A Nisbet, PH Kelly, AJ Davison, M Luján, MF O'Boyle, G Riley, *et al*, in *2015 IEEE International Conference On Robotics and Automation (ICRA)*. Introducing slambench, a performance and accuracy benchmarking methodology for slam (IEEE, Seattle, 2015), pp. 5783–5790
29. A Weinlich, B Keck, H Scherl, M Kowarschik, J Hornegger, in *Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing*. Comparison of high-speed ray casting on GPU using CUDA and OpenGL, vol. 1 (Proceedings of HipHaC'08, Lake Como, 2008), pp. 25–30
30. RS Oliveira, BM Rocha, RM Amorim, FO Campos, W Meira Jr, EM Toledo, RW dos Santos, in *Parallel Processing and Applied Mathematics*. Comparing CUDA, OpenCL and OpenGL implementations of the cardiac monodomain equations (Springer, Torun, 2011), pp. 111–120
31. M Montemerlo, FastSLAM: a factored solution to the simultaneous localization and mapping problem with unknown data association. PhD thesis (2003)
32. M Njiki, A Elouardi, S Bouaziz, O Casula, O Roy, A multi-FPGA architecture-based real-time TFM ultrasound imaging. J. Real-Time Image Process (2016). doi:10.1007/s11554-016-0563-5
33. NVIDIA Tegra K1 Embedded Platform Design Guide. http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/TegraK1_Embedded_DG_v03.pdf. Accessed Jun 2016