OpenCL and OpenGL Implementation of Simultaneous Localization and Mapping Algorithm using High-End GPU

Mohamed Abouzahir[1,*], *Rachid* Latif[2,**], *Mustapha* Ramzi[1,***], and *Mohammed* Sbihi[1,****]

[1]Systems Analysis, Information Processing and Industrial Management Laboratory (LASTIMI), High School of Technology, Mohammed V University in Rabat, Morocco.
[2]Information Systems and Technology Engineering Laboratory (LISTI), National School of Applied Sciences of Agadir, Ibn Zohr University in Agadir, Morocco.

Abstract. Simultaneous Localization And Mapping (SLAM) algorithms are being used in many robotic applications and autonomous navigation systems. The FastSLAM2.0 addresses an issue of the SLAM problem and allows a robot to navigate in an unknown environment. Several works have presented many algorithmic optimizations to reduce the computational complexity of such algorithm. In this paper, a GPGPU (general-purpose computing on graphics processing units) is exploited to achieve a parallel implementation of the FastSLAM2.0. The GPGPU acceleration is done using two different implementations for parallel programming. The first implementation used OpenGL shading language which is based on the characteristics of graphics hardwares. The second implementation used OpenCL which allows hardware acceleration across heterogeneous architectures. We also explored the impact of the two approaches on the the resulting GPGPU implementation. A comparison related to processing-time and localization accuracy is made using a real indoor dataset. Our results show a significant speedup of the GPGPU implementation over a Quad-Core CPU. We show also that, by adopting the same optimization methodology using the two approachs, the OpenCL implementation is faster and suitable for GPGPU accelerated SLAM algorithms.

1 Introduction

The high performance computing communiy has always been focused in the implementation of compute intensive algorithms they developed. The progression of heterogeneous system architecture and tools that support software implementation has enabled the design of complex algorithms that were not possible a few years ago. We have progressively switch from a separate study of the algorithm and the architecture, to a more advanced approach, that take into account both the algorithm and the architecture for an efficient implementation. The Algorithm Architecture Matching (AAM) is a study of the interaction between algorithm and architecture in order to perform an optimized implementation of the algorithm. The AAM reduces the development time and costs of the algorithm implementation. Matching is a reciprocal process of mapping the algorithm and architecture. It must be based on a formalization in system level which unify the algorithms, architectures and implementations, taking into account different constraints (real-time, energy consumption, embeddability ...), and the distributed nature of information to be processed. This requires a better understanding of the problem under study. SLAM algorithms (*Simultaneous Localization And Mapping*) were mostly implemented using rangefinder sensors. Theses sensors are more precise, robust and provide measurements with fewer errors but they are expensive and therefore inaccessible to the public. SLAM using laser sensor has been around for a long time in navigation application. Along with the use of the laser sensors, there were algorithms that were proposed based on vision cameras which are not expensive.

Visual SLAM has received much attention in the computer vision community in the last few years. The SLAM algorithms are computationally intensive. Therefore, there is a general need, in case of embedded systems, to have a platform that allows a software optimization for efficient and scalable implementation. Graphics hardware acceleration are dedicated to perform some tasks faster than what it is possible with a software running on a sequential processing unit. In general purpose hardware such as the CPU, instructions are executed sequentially. To improve performances, hardware accelerations are implemented on a highly parallel architecture to accelerate design of graphics and to handle a huge amounts of data about a complex scene in real time. Since the early days of the their creation, the GPU resources has been accessed as a programmer using graphics software for rendering operation. Once GPU's have shown themselves to be a good target for certain types of high-performance computations [1], GPU's became more attractive for general-purpose computing through graphics APIs such as OpenGL [1]. However, today, the stage seems to be set for a change as

[*]e-mail: mohamed.abouzahir@est.um5.ac.ma
[**]e-mail: r.latif@uiz.ac.ma
[***]e-mail: mustapha.ramzi@est.um5.ac.ma
[****]e-mail: mohammed.sbihi@est.um5.ac.ma

GPGPU research has matured, another recent development has been the adoption of general purpose computing using modern parallel programming language such as OpenCL. Therefore, in this paper the GPGPU implementation focus on the monocular vision FastSLAM2.0. To our knowledge, this is the first work to evaluate the GPGPU implementation of SLAM algorithm using two differents parallel langages. Even if major studies have been made in GPGPU programming during the last years, only few works deals with a GPGPU SLAM implementation taking into account both the algorithm and the architecture. [2] attained a complete end-to-end implementation of FastSLAM2.0 in embedded CPU-FPGA architecture. They adopted a parallel implementation on FPGA and GPU to optimize all block of the system except the image processing task which was ported into one-core of CPU. In addition, they gave a comparative study between FPGA and GPU accelerators, which shows a significant improvement of the SLAM algorithm using FPGA-based heterogeneous architecture. However, the impact of the adopted parallel language is not treated.
[3] presented a full GPGPU implementation for a large scale monocular SLAM using OpenGL Shading language. The optimizations was promising as the algorithm respected the real time constraints. However, the impact of the OpenGL language on the resulting implementation was not studied.
[4] presented an improved MultiCol-SLAM using embedded heterogeneous CPU-GPU architecture. Thus, they attempted to improve and parallelize ORB feature extraction on GPU accelerator. Besides, they proposed a CPU-based multithreading pipelining method to solve CPU and GPU load imbalance. As a result, their proposal improved the frame rate and achieved a higher speedup and low resource utilization than the CPU implementation.
[5] developed a novel VSLAM system named HOOFR SLAM based on a FAST detector and amended FREAK descriptor. Afterward, they adopt the OpenMP multithreading language to implement HOOFR extractor on CPU due to the machine learning optimization, whereas the HOOFR descriptor is ported into GPU accelerator using OpenCL. As a result, their proposal achieved remarkable gains and effective speedup. Again, the OpenCL optimizations techniques are not described in their paper. The organization of this paper is as follows. In section II, a background about simultaneous localization and mapping algorithms is presented with a description of the FastSLAM2.0. We will also present the functional validation of our implementation giving localization results of the algorithm with two different exteroceptives sensors (Laser, Camera) using a real indoor dataset. Since general purpose computing may be unfamiliar to many researchers in the signal processing community, we will give a brief tutorial in Section III, where a background material for GPGPU, using graphic software and a modern parallel programming, are reviewed. Section IV is devoted to the software hardware matching of the FastSLAM2.0 algorithm on a GPGPU architecture. An experimental study is presented comparing performances of the Quad-Core implementation, the OpenCL and the OpenGL GPGPU implementations.

2 FastSLAM2.0: Fast Simultaneous Localization and Mapping

The particle filter-based SLAM algorithm uses a finite set of particles to represent the probability distribution. A high particle density means a high probability of the corresponding area, while a small number of particles means a low probability. The main steps of the algorithm are described as follows:

2.1 Image Processing:

The SLAM algorithm uses visual landmarks to Map the environment and improve robot location estimation. landmark are extracted using FAST feature detector [6]. The SLAM system have to detect previously observed landmarks to correct the map and the location of the robot. The extracted landmark is identified by a simple descriptor, which consists of a window Pixels around the observed landmark. The Matching between observations and landmarks in the map are implemented using zero mean sum of squares Difference ZSSD (1).

$$\sum (I_{lmk}(i,j) - m_d - I_w(x+i, y+i) + m_w) \qquad (1)$$

m_d and m_f are respectively the means of the descriptor pixels and the image window. I is the pixels intensity.

2.2 Prediction

It consists of sampling a new robot's pose at time t given each particle s^m_{t-1}. This is obtained via the probabilistic motion model (2). It calculates the new particle pose by incorporating the odometers data $u_t\,(n_l, n_r)$, where n_l and n_r are respectively the left and the right wheels encoders data [7].

$$s^m_t = s^m_{t-1} + \begin{pmatrix} \delta_s \cos\left(\theta + \frac{\delta_\theta}{2}\right) \\ \delta_s \sin\left(\theta + \frac{\delta_\theta}{2}\right) \\ \delta_\theta \end{pmatrix} \qquad (2)$$

δ_s and δ_θ are respectively the longitudinal and the angular displacements.

2.3 Computing the New Proposal Distribution:

This step merges the most recent measurements into the proposal distribution in order to match it with the posterior distribution. We calculate the new pose from the updated proposal distribution. When the landmark is re-observed, the old proposal distribution from the prediction step is updated. On the other hand, in order to prevent the resampling step from causing more waste to its samples, we calculated based on the observed landmarks, new Gaussian mean and covariance matrix. This Gaussian is built incrementally for each particle and allows a new position to be calculated.

2.4 Estimation:

In the estimation step, if the landmark has been previously observed, the pinhole model (3) is used to predict its position in the image. (u, v) is the position of the landmark in the current image. $(x_{a,cam}, y_{a,cam}, z_{a,cam})$ is the position of the landmark in the scene. We calculate the innovation between observation and prediction. We use EKF's classic equation to update the location of the landmark.

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

c_u, c_v, f_{k_u} and f_{k_v} are the standard camera calibration parameters.

2.5 Resampling:

This Task sample from the proposal distribution and generate new trajectories. The re-sampling step deletes very unlikely trajectories in order to keep the number of particles constant based on the weight of each particle calculated in the estimation step. A new set of particles is drawn and replaced with a probability proportional to the weight. This sampling technique is referred to as importance resampling [8].

2.6 Initialization:

During the mapping process, the SLAM algorithm inputs must be the initial landmark position and the covariance matrix. The initial parameter of the landmark is the inverse depth parameterization [9]. In our work, we use the undelayed method for landmark initialization, because it is more suitable for the particle filter-based SLAM algorithm [10].

The initial coordinates of the 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 angle, φ_i is the elevation angle, and ρ_i Is the inverse depth.

3 GPGPU FastSLAM2.0 Implementation

In this section, we will describe two GPGPU implementation methods and experimental results related to running time.

3.1 Image Processing Task

We have implemented a FAST detector for feature extraction in image processing tasks. The FAST corner detector is a sequential algorithm optimized by machine learning. Therefore, we implemented the FAST detector in the single core of the CPU. The matching process between landmarks and features is based on high-weight particle. Therefore, the matching process is parallelized on a quad-core CPU.

3.2 Prediction Task (FB1)

3.2.1 FB1 OpenGL implementation:

The particle pose $s_t^m = (x, y, \phi)$ is transferred to a texture whose size is equal to the number of particles in the global GPU memory with RGBA internal storage. Therefore, each texel in the texture memory maintains a particle state. For the OpenGL implementation, the random number is generated by the CPU for each particle and then transferred to a separate texture in the GPU memory. This adds a lot of data transfer in each iteration, but using PBO to upload textures is quite fast. In addition, a lot of encoder data is acquired at each time stamp. Therefore, multiple prediction operations are required to correctly reconstruct the trajectory. Therefore, the particle pose must be updated on each received encoder data. To achieve this, multiple rendering passes are required. The only disadvantage is that the texture memory is either read-only or write-only. Therefore, in our implementation, we need three textures for predicting functional blocks: an unchanged read-only texture for the noise encoder data u, and two textures attached to the frame buffer object: one read-only texture is used to input the particle pose s_{old}^t and the write-only texture containing the calculation result s_{new}^t. This means that we swapped the roles of the two textures s_{new}^t and s_{old}^t because we don't need the value calculation in s_{old}^t After the new value is found.

3.2.2 FB1 OpenCL implementation:

First, the particle state (position and covariance matrix) is initialized and saved in a buffer memory in the GPU. To take into account the noisy odometric data, We use the following Gaussian model:

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

$$n_r = n_r + 4 * \left(\sqrt{(-2.0 * \log(\epsilon_3))} * cos(2 * \pi * \epsilon_4) \right) * \epsilon_g \quad (5)$$

- n_l, n_r are the left and right wheels encoders data.
- $(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4)$ are random numbers between [0, 1],
- ϵ_g is the wheels slipping noise

OpenCL randomization process

Each particle is executed by a processing unit (PU), and each PU has its own identity. Therefore, instead of generating random numbers for each particle, we need to build 60 random numbers on the CPU. The generated value is transferred to the GPU. Each random number is combined with the thread ID to generate a private random value for each particle. Then use the thread core to calculate the noise, and then calculate the particle pose and save it in the same memory buffer. Note that this method cannot be implemented in OpenGL because the particles in the texture are accessed using texture coordinates, which are automatically generated by the rasterization pipeline using interpolation. So we can't completely control this value, because it depends on the actual sampling method of the texture.

3.3 Update Proposal Distribution Task (FB2)

3.3.1 FB2 OpenGL implementation:

In order to implement Gaussian construction on the GPU pipeline, multiple rendering passes are required, because the proposal distribution calculated according to the matched landmarks is required as the input of the new one. The average distribution $\mu_t^m = (x, y, \theta)$ and its 3x3 covariance matrix Σ_t^m are calculated in each Rendering passes. We repeat this for each particle in the filter. The fragment shader can only be read from the texture memory, and a texture can only contain 4 parameters (RGBA format). Therefore, we need 4 textures to save the calculated Gaussian and its parameters: 1 texture for the particle position, and 3 textures for the covariance matrix. The problem is that a simple output texture with a size equal to the number of particles stored internally in the RGBA format is not enough to save the calculated Gaussian. Another solution is to write to multiple output textures in one rendering process. The latest graphics hardware supports this feature. The proposed implementation uses four output texture with RGBA format storage. This is enough to save the estimated mean and covariance matrix for each particle in one render pass. We need then, twelve output texture for the Gaussian construction:

- The matched landmark parameters $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$ are saved in four read-only texture.
- Four read-only texture attached to the color attachment of the frame buffer object is used to hold the initial Gaussian mean and covariance $\left(\mu_{old}^{m,t-1}, \Sigma_{old}^{m,t-1}\right)$.
- Four write-only textures attached to the color attachment of the frame buffer object hold the result of one render pass of the updated Gaussian $\left(\mu_{new}^{m,t}, \Sigma_{new}^{m,t}\right)$.

For the next render pass we swap the role of the eight textures attached to the frame buffer object: The four write-only texture will be read-only and the four read-only texture will be write-only. The swap operation is repeated until the Gaussian is constructed completely. This is illustrated in Fig. 1.

3.3.2 FB2 OpenCL implementation:

We implemented the FB2 block in OpenCL as follows: First, we transmit the matched landmark parameters as well as the particle pose and covariance matrix at the same time. Therefore, the Gaussian construction can be completed immediately in the OpenCL kernel. The constructed Gaussian function is stored in the GPU global memory. Please note that for the OpenGL implementation, one execution of the fragment shader will calculate the corresponding Gaussian based on a matching landmark. In order to completely construct Gaussian based on all matched landmarks, in each iteration, we need to transfer the matched landmark parameters to the GPU texture memory. In contrast, in the OpenCL implementation, this is done inside the kernel, so a kernel execution

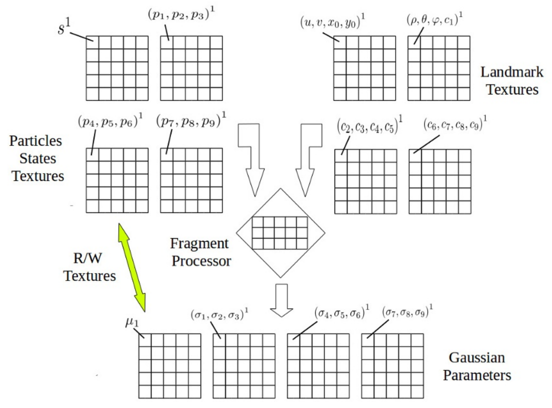

Figure 1. The matched landmark parameters, particle state and the computed Gaussian relative to the first particle (id=1), are stored in the first texel of the four input/output textures. The size of textures is the number of particles

completely builds Gaussian because all matching landmarks are transferred to the GPU global memory before the kernel is executed. Unfortunately, this is not allowed in OpenGL, because as we saw before, a matching landmark parameter occupies 4 textures. More landmarks require more texture allocation, which is impossible for any current OpenGL implementation. 32 is the maximum number of textures that can be allocated even in the latest NVIDIA GPU models. Nevertheless, allocating more texture memory should be avoided. This implementation will increase memory requirements, so the available on-board memory is almost completely occupied by these textures. Shaders and frame buffers, also use on-board memory. Therefore, the OpenCL implementation is efficient, and it allows the Gaussian construction of all particles to be completed in one kernel execution. This reduces the data transfer per iteration (compared to OpenGL), which means a significant improvement.

3.4 Estimation task (FB3)

3.4.1 FB3 OpenGL implementation:

In the OpenGL implementation of the FB3 functional block, we correct one matched landmark using the EKF at each render pass for each particle in a parallel way. To implement the estimation task in OpenGL, we need to allocate twelve textures as follows:

- Eight read-only textures used as inputs to the fragment shader to save the both the old matched landmarks parameters $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$ and the current particles position and covariance matrix (s_t^m, P_t^m) (Fig.1)
- Four write-only textures attached to the frame buffer object to save the new landmark parameters after update for each input particle

The fragment processor execute the EKF filter to corrects the matched landmark parameters and also compute the corresponding likelihood. The estimation results are transferred from the render buffer of the GPU to the CPU since the binary tree is implemented in the CPU side. Typically, at each SLAM iteration, a lot of landmarks are matched with current observation, in this case, we need many iterations to update them all (each iteration process one EKF update to correct one matched landmark for all particle). Multiple iteration causes significant data transfer between output textures and CPU, but texture transfer is rather faster when using PBO.

3.4.2 FB3 OpenCL implementation:

The estimation functional block FB3 is implemented in efficient way in OpenCL then the OpenGL implementation. We mentioned earlier in FB2 OpenCL implementation, that all the matched landmarks parameters are all transferred to the GPU global memory, therefore there is no transfer that is done before and after kernel execution. So, we need only to trigger the kernel execution that implements the extended kalman filters to update all the matched observation parameters in one kernel execution. Compared with the OpenGL implementation, the data transfer before and after the fragment shader is executed is inevitable. As mentioned earlier, texture memory is a very limited resource. We cannot use more than 32 textures to save the state of all matching landmarks. Therefore, the realization of the estimation function block in OpenCL has a significant improvement over OpenGL.

3.5 Inverse depth initialization (FB4)

3.5.1 FB4 OpenGL implementation

As mentioned above, for the landmark to be initialized for all particles in the filter, five parameters must be calculated. First, the inverse depth parameter ρ has a constant initial value $\rho = 0.25$ [9], so it is initialized in the CPU. However, the initial camera pose, azimuth and elevation $(x_i, y_i, \theta_i, \phi_i)$ are calculated in the graphic hardware. In order to calculate this parameter, GPGPU first needs the position of the landmark and the position of the particle in the current frame. Therefore, the initialization function block only needs two textures: a read-only texture with RGBA internal format, used to save particle poses, one particle per texel $\left(R = x_p, G = y_p, B = \theta_p, A = 0\right)$, a write-only texture is attached to a framebuffer object that saves the initial landmark parameters, and each pixel has a landmark $(R = x_i, G = y_i, B = \theta_i, A = \phi_i)$. The landmark pose in the image (u_i, v_i) is loaded into the shader by the API as a uniform value. The kernel executes all particles in parallel to calculate four initialization parameters. In order to perform the calculations and ensure that our fragment shader is executed for all input particles at once, a filled quad is drawn. We choose the size of the quad so that there is exactly one fragment per particle. The rendering result is retained on the output texture of the frame buffer object attached to the color attachment. The result is transmitted back to the CPU via PBO to speed up the transmission.

3.5.2 FB4 OpenCL implementation

Inverse depth initialization (FB4) is implemented differently. We also transmit the matched landmark parameters and their corresponding observation parameters in the particle update block (FB2). Therefore, the kernel is executed directly without any previous transfers. The kernel computes the inverse depth parameterization for the newly observed landmark and all particles in parallel. As we mentioned before, we have implemented a binary tree for map management. Therefore, we transfer all updated matching landmarks (updated in FB3) and initialized landmarks (initialized in FB4) back to the CPU. On the other hand, in the OpenGL implementation of the FB4 block, we only transfer the newly initialized landmarks to the CPU. Because in the FB3 OpenGL implementation, all matching landmarks have been transferred back to the CPU. This allows the OpenGL implementation to be more efficient for this block parallel implementation.

3.6 Resampling task (FB5)

The resampling function block replaces low probability weight particles with higher weight particles to prevent the filter from being exhausted. Generate a new set of particles according to the corresponding probability weights. The most time-consuming subtasks in the resampling block are implemented in parallel on the GPU. The resampling block contains two main tasks: resampling the particle state, which is implemented in parallel on the GPU, and resampling the particle map landmarks on the CPU, because the mapping is implemented as a binary tree in the CPU. The total weight calculation and resampling threshold (N_{eff}) is a time-consuming process:

$$N_{eff} = 1/\left(\sum_{i=1}^{M} \omega_i^2\right) \tag{6}$$

(ω_i is the normalized weight)

In order to achieve the weight summation, we use a reduction operation method in OpenGL. The reduction operation maps the MxM texture to the 1x1 texture. This operation recursively reduces the output texture size by calculating the local sum of each group of 2x2 elements in a kernel and writing it to the corresponding output location. This operation is called building a mipmap pyramid in computer graphics. We adjust the texture coordinates used to calculate the input texture to read the $2x2$ block. For this, we use the multi-texture coordinates supported by OpenGL. We assign multiple texture coordinates to the vertices of the drawn quadrilateral. The rasterizer inserts coordinates on the quadrilateral. Therefore, in the fragment kernel, we only input the interpolated coordinates and use them to sample the input texture four times.

3.6.1 Weight summation implementation

The sum operation is divided into K sub-tasks. Each subtask takes the sum of all particles weights with the same surplus. The corresponding identification is divided by K.

The resulting total weight is the sump of all sub-task outputs. Each individual weight sub-tasks are processed in parallel in separate GPU kernels. The results of sub-tasks are transferred to the CPU that calculate the total weight. To allow such implementation, the number of sub-tasks must be a multiple of the number of particles.

3.6.2 computation of the minimum number of effective particles

We use the same implementation method as the weight summation. A reduction operation is performed in a ping-pong manner: Each sub-tasks compute the square sum. We used three textures: One input texture hold the input weights w and two output texture to perform the reduction operation. Rather with OpenCL, each subtasks compute the square sum of a group of particles weights. The resulting final value is done on the CPU side.

4 Results and Discussion

In order to evaluate the parallel implementation discussed above, we use the Rawseeds dataset collected in the indoor environment [**?**] to run and time evaluate FastSLAM2.0's OpenCL and OpenGL GPGPU implementations. The comparison is made using wheel encoder and monocular camera data. The algorithm runs and evaluates time for different numbers of particles. Fig.2 gives a quantitative overview of OpenCL and OpenGL GPGPU implementation and their influence on different functional blocks of the algorithm.
Table 2 shows the execution time of the GPGPU implementation of the function block when the FastSLAM2.0 algorithm is run for 100 time steps, using different numbers of particles range from 2^4 to 2^{12}. Let us recall that for the prediction step (FB1), we used two different implementations to generate the noise used in the probabilistic motion model. When $N = 256, 1024, 4096$, the OpenCL implementation of the prediction step is faster than OpenGL.
In these cases, we transfer 60 random numbers from the CPU to the GPU, and for the OpenGL implementation, we transfer the random numbers generated for all particles in the filter. This is shown in an implementation with 16 particles, where the transmission of 16 random numbers is faster than the transmission of 60 numbers, so in this case, the OpenGL implementation is faster.
In the particle update function block (FB2), for OpenGL and OpenCL implementations, we transfer the matching landmarks, matching indexes, and observations made from the CPU to the GPU for all particles. Although the load of the two implementations is the same, in most cases, the (FB2) GPGPU implementation using OpenCL is faster than OpenGL. In fact, the Gaussian construction using OpenCL is completed in the computing kernel. All matching landmarks are saved in GPU memory. Compared with OpenGL, texture memory is a limited resource and cannot accommodate more than 32 allocated textures. This makes it necessary to transfer each matched landmark to the GPU before the fragment shader is executed. Therefore, the performance is reduced compared to the OpenCL implementation.
The calculation time results obtained in the estimation function block are expected. In fact, as mentioned earlier, in the OpenCL implementation, before and after the kernel is executed, the block is executed without any transmission. This is because the matched landmark parameters have been transmitted in (FB2). On the other hand, in the OpenGL implementation, transmission is necessary because texture memory is limited. So, the GPGPU implementation with OpenCL is rather faster.
In contrast, for a large number of particles, using OpenGL to achieve inverse depth initialization (FB4) is faster. In fact, in the OpenCL implementation, we pass the entire map particles (updated and initialized landmarks) from the GPU back to the CPU in order to arrange them in a binary tree. For the OpenGL implementation, this is done in two different stages (matching landmarks are transferred back to the CPU in FB3, and new landmarks in FB4), which reduces the transfer cost. In the last function block (FB5), we implemented the same optimization strategy for OpenCL and OpenGL, although GPGPU using OpenCL is faster.
In terms of global processing time, the results of (Fig.3) show that for all numbers of particles, the GPGPU implementation using OpenCL is faster than OpenGL. The table 1 shows a comparison of the execution time (100 time steps) between the quad-core and GPGPU implementation. The result (Tab.1) shows that for OpenCL and OpenGL, the GPGPU implementation runs faster than the quad-core implementation. In fact, for OpenGL, our GPGPU-based implementation achieves more than 15 times the speedup than the quad-core implementation. OpenCL is released 18 times faster than the quad-core implementation. This comparison is achieved for 4096 particles, FastSLAM2.0 can provide more accurate mapping and positioning results. Figure 4 shows the global acceleration based on the quad-core implementation for OpenCL and OpenGL GPGPU. When the number of particles is increased , the global acceleration with OpenCL increases substantially over the OpenGL one.
Fig.5 shows the evolution of CPP (period per particle) [11] for different implementations. This metric is calculated using the following equation: $CPP = \frac{f*t}{M}$ where M is the number of particles, f is the clock frequency represented by Hz, and t is the processing time in seconds. As long as the number of particles increases, the CPP for the GPGPU implementation will maintain a lower value than the quad-core implementation.

4.1 Synthetic Results

Given that GPU architectures are inherently dedicated to graphics workloads, are they suitable for optimizing computationally intensive algorithms, such as SLAM problems? Which language is suitable for extracting the maximum performance benefit from the GPU for non-graphic workloads? Is it a graphical method because it is the first language to program the GPU, or is it a new modern

N	16			256		
FBs	OpenMP	OpenGL	OpenCL	OpenMP	OpenGL	OpenCL
FB1	35.75	35.72	50	463.22	50.9	40
FB2	61	90.72	10	488.23	127.96	130
FB3	215.9	162	20	1160	207.86	50
FB4	56.9	36.83	30	202.12	39.96	140
FB5	5.01	57.76	10	11.72	32.02	10
Total (ms)	374.56	383.03	120	2325.29	458.7	370
N	1024			4096		
FBs	OpenMP	OpenGL	OpenCL	OpenMP	OpenGL	OpenCL
FB1	1591	80	50	6371	125.25	90
FB2	1671	164	100	2790	269.11	270
FB3	4344	290.81	90	6884	542.63	150
FB4	272	6.5	170	120	13.55	330
FB5	98	90	30	75.65	67.56	40
Total (ms)	7976	631.31	440	16240.65	1018.1	880

Table 1. FBs processing times (ms) of the FastSLAM2.0 for 100 time-steps

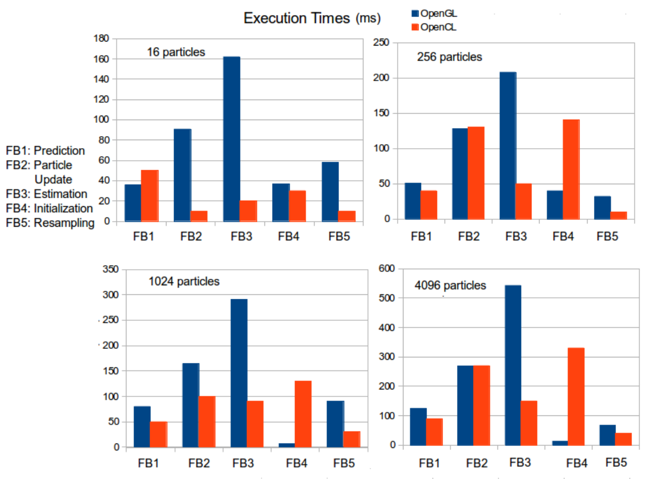

Figure 2. Processing time comparison of the different functional blocks between the GPGPU implementations using OpenCL, OpenGL

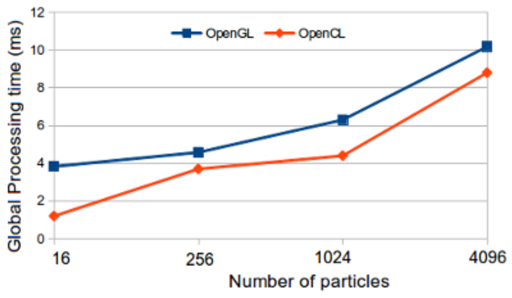

Figure 3. Global processing time comparison of the GPGPU implementation between OpenCL, OpenGL

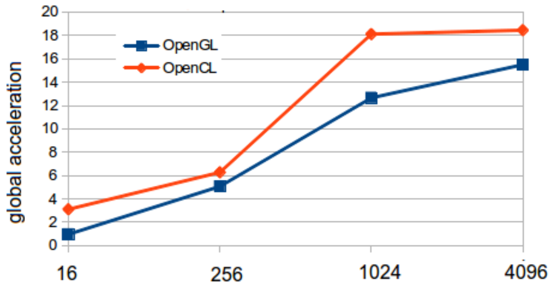

Figure 4. Global acceleration comparison of the GPGPU implementation between OpenCL, OpenGL

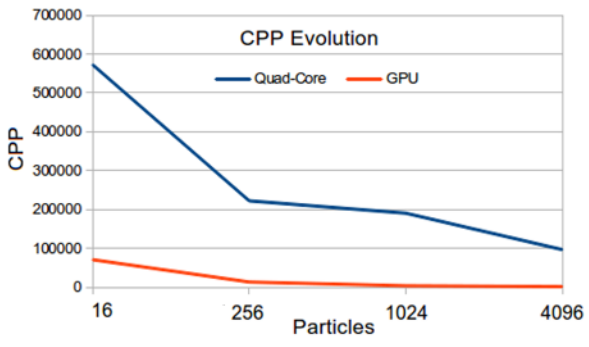

Figure 5. CPP evolution

language dedicated to parallel programming? To answer this question, we implemented FastSLAM2.0 as an algorithm worthy of autonomous robot navigation on high-end GPUs using two GPGPU languages. The results obtained led us to conclude that GPUs provide performance advantages for optimizing SLAM algorithms. In addition, our research led us to conclude that we can achieve higher acceleration using OpenCL, which provides better memory management than OpenGL. We can freely access the local, global and constant memory in the OpenCL kernel. This allows us to better manage the data transmission and data structure of FastSLAM2.0. OpenGL provides three types of memory: vertex buffer, texture, and frame buffer. The fragment shader can only use texture memory for GPGPU purposes. The limited architecture of this memory has an impact on the final OpenGL implementation. We need to transfer input/output data to and from the CPU in each iteration. In addition, the frame buffer is a write-only memory. Once written, the fragment shader cannot read data from this memory. This makes this memory useless for GPGPU. Therefore, any GPGPU purpose requires the use of frame buffer objects. As mentioned earlier, the frame buffer object allows the use of texture memory to save fragment shader output and use them as input (multi-render pass and ping-pong technique). Therefore, read/write operations are done in texture memory. In fact, this memory is limited, and the performance of GPGPU will also be limited. This is an experiment conducted while exploring the implementation of FastSLAM2.0 using OpenGL.

5 Conclusion

As far as we know, this is the first work to evaluate GPGPU Implementation of FastSLAM 2.0. We used OpenGL shading GLSL language and open computing language OpenCL, dedicated to Parallel programming. This choice is based on the fact that most Of current GPUs support these languages, unlike CUDA which is specific to NVIDIA hardware. First, use real indoor data sets to evaluate the consistency of the algorithm. The algorithm was tested in two indoor environments using two external sensor data (laser and camera). Then the complete realization of monocular FastSLAM2.0 is described. The image processing task is implemented on the CPU, and the FastSLAM2.0 algorithm is mapped on the GPGPU. We took advantage of some OpenGL/OpenCL features to speed up processing time and reduce data transfer costs. Development has shown that OpenGL is difficult to handle because it requires prior experience in graphics programming, underlying graphics pipeline hardware, and has some memory management issues. OpenCL provides an efficient parallel programming that can achieve high performance. In fact, compared to the quad-core implementation, the overall processing time is accelerated by 18 times. The results show that the general-purpose GPGPU implementation can greatly accelerate the synchronization positioning and mapping algorithms of real-time applications, such as FastSLAM2.0.

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, *A Survey of general-purpose computation on graphics hardware*, in *Computer graphics forum* (Wiley Online Library, 2007), Vol. 26, pp. 80–113

[2] M. Abouzahir, A. Elouardi, R. Latif, S. Bouaziz, A. Tajer, Robotics and Autonomous Systems **100**, 14 (2018)

[3] M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, A. Tajer, EURASIP Journal on Advances in Signal Processing **2016**, 88 (2016)

[4] J. Li, G. Deng, W. Zhang, C. Zhang, F. Wang, Y. Liu, Journal of Real-Time Image Processing **17**, 713 (2020)

[5] D.D. Nguyen, A. Elouardi, S.A.R. Florez, S. Bouaziz, IEEE Transactions on Intelligent Transportation Systems **20**, 4103 (2018)

[6] E. Rosten, R. Porter, T. Drummond, IEEE Trans on Pattern Analysis and Machine Intelligence pp. 105–119 (2009)

[7] E. Seignez, M. Kieffer, A. Lambert, E. Walter, T. Maurin, IEEE Int. Journal of Robotics Research **28**, 34 (2009)

[8] W.G. Madow, L.H. Madow, The Annals of Mathematical Statistics **15**, 1 (1944)

[9] J. Montiel, J. Civera, A. Davison, *Unified inverse depth parametrization for monocular SLAM*, in *Int. Conf. on Robotics: Science and Systems* (2006), pp. 16–19

[10] E. Eade, T. Drummond, *Scalable Monocular SLAM*, in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2006), Vol. 1, pp. 469–476, ISSN 1063-6919

[11] M. Njiki, A. Elouardi, S. Bouaziz, O. Casula, O. Roy, *A real-time implementation of the Total Focusing Method for rapid and precise diagnostic in non destructive evaluation*, in *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)* (2013), pp. 245–248