Fast Processing of Grid Maps using Graphical Multiprocessors

Diego Rodriguez-Losada, Paloma de la Puente, Alberto Valero, Pablo San Segundo, Miguel Hernando

Centro de Automatica y Robotica (UPM-CSIC)
Madrid, Spain. Tel: 913367729 e-mail: diego.rlosada@upm.es

Abstract: Grid mapping is a very common technique used in mobile robotics to build a continuous 2D representation of the environment useful for navigation purposes. Although its computation is quite simple and fast, this algorithm uses the hypothesis of a known robot pose. In practice, this can require the re-computation of the map when the estimated robot poses change, as when a loop closure is detected. This paper presents a parallelization of a reference implementation of the grid mapping algorithm, which is suitable to be fully run on a graphics card showing huge processing speedups (up to 50X) while fully releasing the main processor, which can be very useful for many Simultaneous Localization and Mapping algorithms.

Keywords: Mobile robots, Robot navigation, Computer Graphics

1. INTRODUCTION

Although microprocessor manufacturing technology is continuously improving, it is reaching the point in which physical limits are becoming a major concern. Memory speed and power have imposed walls for increasing processing performance by scaling the clock frequency. Over the last years, Moore's law and performance improvements have been maintained mainly due to one reason: multi-core processors (multiprocessors). In multiprocessors, several CPU cores are packaged into a single chip, taking advantage of their proximity, for example when accessing the cache memory. Some well known examples are Intel Dual-Core and Quad systems, Sony Cell (8-core) processor inside the PlayStation3 and PowerPC Xenon (3-core) processor of the Microsoft Xbox 360.

Together with multiprocessors, programming models have been coming to scene, in order to manage and exploit the available concurrency in those systems. Message Passing Interface (MPI) is the *de facto* standard for high performance in distributed computing (multiple external processors each one with its own memory), while OpenMP is probably the most extended solution for multiprocessing in shared memory systems, as multi-core CPUs.

Manufacturers of graphical processing units have been also continuously improving their systems, leading to multi-core Graphical Processing Units (GPUs) in which each core contains also a large number of Arithmetic and Logical Units (ALUs) specialized in parallel processing of graphics as textures, visibility, image processing, etc. Also the large market for graphics cards with ubiquitous 3D graphics (games, CAD, multimedia, etc), has implied a reasonable cost of very powerful devices that can fit into the class of what is known as commodity hardware. Major GPU manufacturers have recently released tools and programming models that allow programmers to access such computing power: ATI (now part of AMD) development platform is called ATI Stream, and the Nvidia one is called Computed Unified Device Arquitecture (CUDA).

The CUDA approach has gained large attraction and many researchers have found a powerful computing platform for boosting the computations required in their job. Furthermore, several libraries as CUBlas (a port of the Basic Linear Algebra Set – Blas) or GpuCV (largely compatible with OpenCV) for computer vision have been developed that let developers take advantage of GPUs computing power without requiring explicit parallelization of algorithms. Applications such as Matlab or GIMP have also been provided with CUDA extensions that let the applications transparently benefit from GPUs processing.

Many algorithms in mobile robotics are quite computationally intensive. Among them, the map building problem (see Thrun (2008) for a survey) has gained great attention in the last decade. Many solutions have been developed, as feature based Simultaneous Localization and Mapping (SLAM) with Extended Kalman Filters (EKF) and Information Filters, and particle filter FastSLAM. Feature based representations have the disadvantage of rejecting observations that cannot be modelled as a priori known features, typically lacking enough information to safely perform path planning or control tasks. Probabilistic grid maps (Elfes, 1987) divide the environment into small square cells and compute the probability of each cell to be occupied or free, independently from the other cells, so in order to obtain a reasonable result, the robot poses have to be known (mapping with known poses). Although another higher level approach has to be used for obtaining these poses, the final computation of the grid map is always a necessary task.

In this paper we propose a parallelization of the grid mapping algorithm able to run on a multi-core GPU. The enormous speedup obtained shows the effectiveness of our approach,

and the potential of the system, that could open the door to a new perspective about robotic algorithms.

The rest of the paper is structured as follows: First, previous related work is discussed in Section 2. A brief introduction to CUDA is provided in Section 3, and the common grid mapping algorithm is described in Section 4. The proposed parallelization of grid mapping is explained in Section 5, the obtained results with such approach are presented in Section 6, and the conclusions are exposed in Section 7.

2. RELATED WORK

There exists a very closely related work from Yguel et al. (2008), in which GPU based processing of grid maps is also presented. Their work claims that grid mapping algorithms based on ray tracing, commonly using the line tracing algorithm of Bresenham (1965) are inaccurate as they present the *de Moire* effect, defined in the computational graphics domain as artificial discontinuities between rays far from the origin. They propose a so called exact solution and derive a thorough probabilistic formulation with the hypothesis that the unmodified cells between rays should be also updated accordingly. With this hypothesis they conclude with an algorithm that can be efficiently run in a GPU with the application of typical texture handling functions. Although their derivation is an important contribution it is only applicable in the case in which the laser beam width angle is equal to the angular spacing between consecutive beams, which is not a valid hypothesis in the general case of a typical laser range-finder. It is easy to validate this statement with a simple experiment as presented in Fig. 1. A SICK laser LMS200 is used, with an angular resolution of 1º, and an object of approximately 5 cm wide (double than the cell size we use in our experiments) is moved in front of the laser at approximately 4 meters. The object continuously appears and disappears while crossing the beams and the non-covered inter beams space respectively.

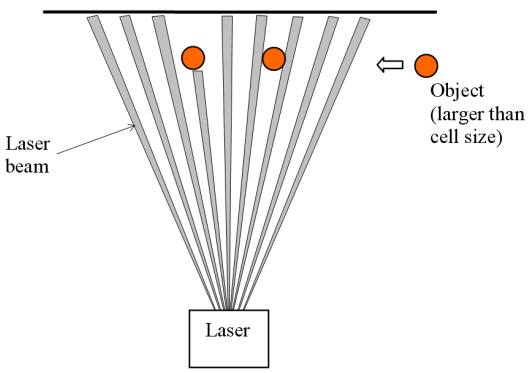

Fig. 1. Counter example to the exact solution proposed in Yguel (2008) The beams width is not equal to the inter beam space.

This seems a reasonable result as the approximate beam angle is about 0,25º (as derived from the manufacturer technical data). At the light of this result, it is reasonable to follow a Bresenham line tracing algorithm, that might be not as accurate as a beam model, but in any case it is safer than a model with a wider than real laser beam.

The work presented in this paper follows a different approach: it takes a well known algorithm and an extended implementation as the one in the CARMEN toolkit, and implements it for running in a GPU, by finding and adequate parallelization. Furthermore, the full grid mapping algorithm is addressed to the GPU, as opposed to Yguel et al. (2008) in which only certain parts of the algorithm are processed in the GPU, so we are able to present absolute timings comparison between the same algorithm in a CPU implementation and a GPU one. The main contribution of the paper is the proof that exactly the same algorithm that is being used by a large part of the community can be targeted to a graphical card with large computational savings and fully releasing the CPU.

3. OVERVIEW OF NVIDIA CUDA ARCHITECTURE

CUDA exposes the Nvidia multi-core GPUs computing capabilities through the following elements:

- Thread hierarchy. The programmer develops a function of a working thread which can be massively launched in a kernel. The threads can be arranged in blocks, each block containing the same arrangement of threads. Such arrangement can get the form of a vector (1D), a matrix (2D), or a generic 3D matrix. The blocks themselves can also be arranged in vector or matrix forms in a so called grid. Each thread has built-in variables to access its indices in the block, as well as the block indices.

- Memory hierarchy. Each thread has its own private memory space, each block has a shared memory that can be accessed by all threads in the block, and there exists a global memory accessible by all threads. The system is completed with two read-only memory spaces: the constant memory and the texture memory. The shared memory is built inside the GPU, so it is faster than the global memory that is outside the GPU (but located in the device, i.e. the graphics card).

- Thread synchronization. All threads in a block can be forced to wait at a given point until all threads of the block arrive to that point.

The use of these elements is available to the programmer via some extensions of the C language as well as a runtime library. With these extensions the programmer can define kernels, declare the type of device memory for each data, and synchronize threads. The framework has a low learning curve, but it should be said that optimization typically requires some extra work.

A typical working cycle when using GPU consists of the following steps: allocating memory on the device, copying data from host (PC) memory to the device, launching a kernel, and finally copying the results from the device memory to host memory.

In order to obtain a good performance several things have to be considered. The threads have practically no overhead of execution, changing, and finishing. The memory accesses to arithmetic computations ratio has to be low, i.e. the memory is quite slow compared with the computational capabilities. Also it has to be considered that shared memory is much faster than global memory. However, the memory latency can be typically hidden if there are enough threads to be scheduled. In practice this means that a kernel must launch thousands of threads (all of them running the same code) in order to achieve a good performance.

4. BUILDING GRID MAPS WITH MOBILE ROBOTS

The problem of estimating a map m given the set of sensor measurements z^t and robot poses x^t up to timestep t in probabilistic terms is represented by the posterior $p(m \mid z^t, x^t)$. This conditioning on the robot poses is known as *mapping with known poses*. The sensor measurement and robot pose at a certain time step i will be grouped (for convenience) into a sensor interpretation called s_i. In 2D grid mapping, the environment map m is divided in multiple square cells arranged in a grid fashion, being $m_{x,y} \triangleq m_c$ the cell c with indices (row and column) x and y. To cope with the high dimensionality of the problem of estimating the posterior $p(m \mid s^t)$, the problem is simplified assuming that the probability of occupancy of each individual cell is independent from each other:

$$p\left(m \mid s^t\right) = \prod_c p\left(m_c \mid s^t\right) \tag{1}$$

As explained in Thrun (1998), the probability of each cell given all the sensor information can be computed as:

$$\begin{aligned} p(m_c \mid s^t) &= 1 - \left(1 + \frac{p_{prior}(m_c)}{1 - p_{prior}(m_c)} p'\right)^{-1} \\ p' &= \prod_{i=1}^{t} \frac{p(m_c \mid s_i)}{1 - p(m_c \mid s_i)} \frac{1 - p_{prior}(m_c)}{p_{prior}(m_c)} \end{aligned} \tag{2}$$

Where $p_{prior}(m_c)$ denotes the prior probability of occupation, and it is an initial parameter of the algorithm. It is assumed to be equal for all cells and thus can be simplified to p_{prior}. The above expression (2) can be conveniently expressed incrementally:

$$p(m_c \mid s^t) = 1 - \left(1 + \frac{p(m_c \mid s^{t-1})}{1 - p(m_c \mid s^{t-1})} \frac{p(m_c \mid s_t)}{1 - p(m_c \mid s_t)} \frac{1 - p_{prior}}{p_{prior}}\right)^{-1} \tag{3}$$

The probability $p(m_c \mid s_t)$ of occupancy of a cell conditioned only on one observation at a certain time step t is established by the probabilistic sensor model. The model handles each laser beam separately, and for each beam uses the Bresenham (1995) line algorithm to define which cells will be affected by the sensor observations. All the cells that are not affected do not have to be updated by (3).

The Bresenham algorithm takes as input a segment defined by two extreme cells: the first one c_1 is the origin of the beam i of the laser scan, and the second one c_2 is computed from the range measurement and an input parameter called $wall_{size}$ that represents the most likely width of the objects of the environment (Fig. 2).

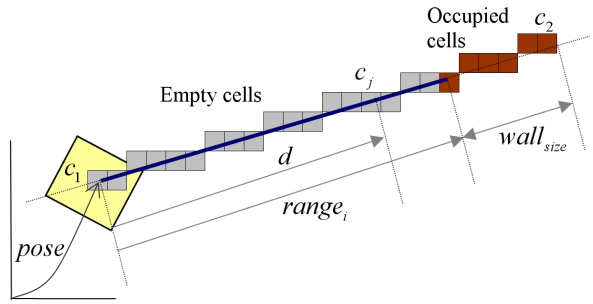

Fig. 2. Bresenham line algorithm for a laser beam. Note that the occupied cells include those located further than the range measurement up to a certain limit $wall_{size}$ that is defined as an input parameter to the algorithm.

The obtained set of cells **C** is thus naturally clustered in two groups, those cells considered to be empty, and those considered to be occupied. Two probabilities are defined as parameters, a reference probability that a cell is empty p_{emp} for those cells that are between the beam origin and the range measurement, and the reference probability that a cell is occupied p_{occ} for those cells beyond the range measurement, up to the wall size.

$$p_f = \begin{cases} p_{emp} & d < range \\ p_{occ} & d \geq range \end{cases} \tag{4}$$

Then the distance from each cell to the origin is computed, in order to be used in the sensor model. The probability function defined by the sensor model is divided in two parts. If the cell is very close to the laser, i.e. below a threshold called $range_{sure}$, then the probability of the cell is directly defined by the corresponding (occupied or empty) reference probability. If the cell distance is above that threshold, the corresponding reference probability varies linearly with the distance. Formally:

$$p(m_c \mid s_t) = p_f + \delta \frac{(d - range_{sure})}{range_{max}} (p_{prior} - p_f) \tag{5}$$

where

$$\delta = \begin{cases} 0 & d < range_{sure} \\ 1 & d \geq range_{sure} \end{cases} \tag{6}$$

The algorithm iterates over all range measurements of the laser scan captured at time step t, and for each range

measurement iterates over the set of cells defined by the Bresenham algorithm.

The first step is to check whether this cell actually belongs to the grid map, as it is possible that range measurements could fall out of the map. Then the sensor model is applied to compute $p(m_c \mid s_t)$ according to (4)-(6), and finally the value of the cell is updated as defined in (3). The outline of the algorithm is:

Function Update(Grid $p(m \mid s^{t-1})$, Scan s_t, Pose $pose_t$)
foreach $range_i \in s_t$
  c_1, c_2=ComputeExtremePoints($range_i$, i, $pose_t$)
  C=BresenhamLineTrace(c_1, c_2)
  foreach $c_j \in \mathbf{C}$
    if($c_j \subset m$)
      d=Distance(c_j, c_1)
      $p(m_{c_j} \mid s_t) = \text{SensorModel}(d, range_i)$
      $p(m_{c_j} \mid s^t) \Leftarrow p(m_{c_j} \mid s_t), p(m_{c_j} \mid s^{t-1})$ (Eq. 2)
    endif
  endfor
endfor

5. PARALLELIZATION OF GRID MAPPING

5.1 Threads arrangement

The core idea of the proposed parallelization is that one GPU thread will be launched to update each cell affected by a sensor observation. As cells are obtained from the Bresenham algorithm for each laser beam, it is adequate to group all the threads corresponding to those cells in the same thread block. Thus, a thread block per range measurement will be used

Inside each block we theoretically need a variable number of threads depending on the actual range measurement, since a variable number of cells should be affected. Nevertheless, CUDA does not allow a variable number of threads in each block, so the number of threads is chosen to fit a maximum range of the measurements. Although the number of threads per block is limited depending on the hardware platform, it starts in 256 threads which can accommodate a sensor range of 6.4 meters for a cell size of 2,5 cm and 12,8 m for a cell size of 5 cm, which are reasonable values.

The proposed kernel will be typically composed of 360 blocks, each one with 256 threads, i.e. a total of 92160 threads for handling each observation. To process a whole map, a kernel is launched for each observation.

The working cycle is as follows: The first step is to allocate and copy the input data to the device memory. In this case it is necessary to copy the whole laser data (including all measurements from all time steps, as well as the robot poses), the probability grid, and the input parameters. Then a kernel is launched for every time step, but with the data already stored in the device. In this way, the transfer of data is avoided at each kernel launch. Please note that the map is not transferred to the GPU at each step. When all data has been processed, the resulting updated probability grid is transferred back to the host memory.

5.2 Kernel computation

Each one of the running threads will have three basic parameters: (I) the time step t which determines which observation has to be processed, (II) the index of the block i, which represents the laser beam index, and (III) the index of the thread j within the block, that represents a cell along the Bresenham line of the beam.

The first step is to compute the extreme points of the laser beam, and also the relative increments between them:

$$\begin{aligned} c_1 &= \{x_1, y_1\}, c_2 = \{x_2, y_2\} \\ \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \end{aligned} \tag{7}$$

According to the Bresenham algorithm the slope of the line has to be lower than 1, so if the absolute value of Δx is larger than the absolute value of Δy, then it is clear that we can compute the position of the cell according to:

$$\begin{aligned} g &= \Delta y / \Delta x \\ x &= x_1 + \operatorname{sgn}(\Delta x)\, j \\ y &= y_1 + g(x - x_1) \end{aligned} \tag{8}$$

Otherwise, the X and Y indices have to be swapped, and the position of the cell is computed by:

$$\begin{aligned} g &= \Delta x / \Delta y \\ y &= y_1 + \operatorname{sgn}(\Delta y)\, j \\ x &= x_1 + g(y - y_1) \end{aligned} \tag{9}$$

Once the cell position is computed, the application of the sensor model is also done with (4)-(6) and the update of the cell probability with (3).

5.3 Memory use

Although the basic computations have been presented, the critical decisions in the parallelization are about memory use.

There are some parameters of the algorithm that are not modified and are accessed many times, as $range_{sure}$, $range_{max}$, p_{emp}, p_{occ}, p_{prior}, $wall_{size}$. These parameters can be conveniently stored in read-only constant memory which is known to be faster than global memory.

The CPU implementation uses a lookup table to precompute and store all the distances from a cell to the origin to avoid recomputing these distances at each time step. As explained

above, the GPUs have a huge computing power compared with memory bandwidth, so the GPU implementation recomputes the distances at every time step avoiding in this way extra memory accesses.

Each block manages a laser beam, so the computation of the extreme points of the beam can be shared among all threads in the block. Thus, the range measurement, the extreme points, and the increments are stored in shared memory. Although they could be recomputed by each thread, it is faster to use shared memory and let one thread compute these values for all threads in the block. A synchronization point is introduced here to make the other threads wait until the first one finishes computing these values.

The grid map probability matrix is stored in global memory. As the computation is done per cell, and only one read and one write accesses are required, the use of shared memory is not justified.

The final parallelized algorithm is outlined as:

Grid $p(m \mid s^{t-1})$, Scan s_t, Pose $pose_t$ //Device Memory
Thread Update(Block i, Thread j) //360x256 threads
  if $(j = 0)$
    $range_i = s_t[i]$
    c_1, c_2=ComputeExtremePoints($range_i$, i, $pose_t$)
  endif
  c_j=BresenhamPoint(c_1, c_2, j)
  if($c_j \subset m$)
    d=Distance(c_j, c_1)
    $p(m_{c_j} \mid s_t) = \text{SensorModel}(d, range_i)$
    $p(m_{c_j} \mid s^t) \Leftarrow p(m_{c_j} \mid s_t), p(m_{c_j} \mid s^{t-1})$ (Eq. 2)
  endif

6. EXPERIMENTS AND RESULTS

We have implemented the presented parallelized approach with the CUDA programming tools, as well as an adaptation of the algorithm developed in CARMEN which is basically a C++ wrapper and some minor modifications in order to be able to make a fair comparison between both implementations. For example, most NVIDIA GPUs do not handle double precision arithmetic, and all doubles are automatically demoted to single precision floats. The original CPU implementation had double precision computations which are typically slower, so all the types were changed to faster single precision. In both implementations all the input data (laser scans and poses) is loaded in memory before starting the computation to eliminate delays from reading data from a hard drive.

Two real different datasets with the robot poses already corrected (see Acknowledgment) have been used for the experiments and processed with two different PCs with Intel CPUs and NVIDIA GPUs. The desktop PC is equipped with an Intel 3,2Ghz Pentium D CPU and a NVIDIA GTX 280 graphics card. The laptop has an Intel 2Ghz Core 2 Duo T7250 and a GF 8400M GS graphics card.

In the first one, a Pioneer2 with a LMS-Laser was manually driven in the building 079 of the University of Freiburg (Fr079), acquiring 3118 lasers scans (each one with 360 beams in a 180º FOV with 0.5º spacing) along a trajectory of approximately 209 meters. The maximum range used is 6.4m.

Table I shows the absolute processing time in seconds required for processing this data set. The processing time is reduced down to 50% (2X) with the laptop graphics card (compared with the laptop CPU), and down to 1,7% (58X) in the desktop PC. For the GPUs, the memory transfer times to and from the graphics card are also included in the results.

TABLE I PROCESSING TIMES (SECONDS)

	ENVIRONMENT	
PROCESSOR	Fr079	Fr101
2Ghz Core 2 Duo T7250	11,5	25,3
3,2Ghz Pentium D	15,2	30,9
GF 8400M GS (laptop)	6,26	12,17
GTX 280 (desktop)	0,26	0,52

Fig. 5 shows the result of processing this data set. Although the map shown in the figure is the one processed with the GTX 280 graphics card, the result is visually identical (as expected) to the one obtained with CPUs.

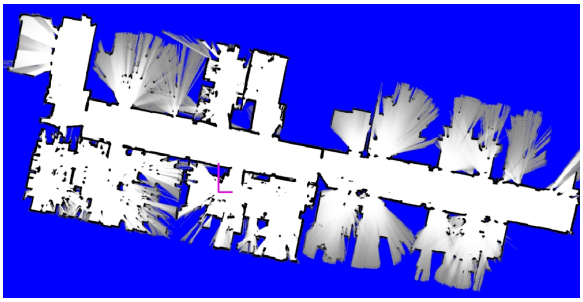

Fig. 5. FR079 building map, computed in 0,26secs with GPU

The second data set was captured in the Entrance Hall (Building 101) at the Department of Computer Science at University of Freiburg, with a Pioneer 2 DX8 robot and with the same laser configuration, during autonomous exploration and capturing 5299 laser scans along a 277 meters trajectory.

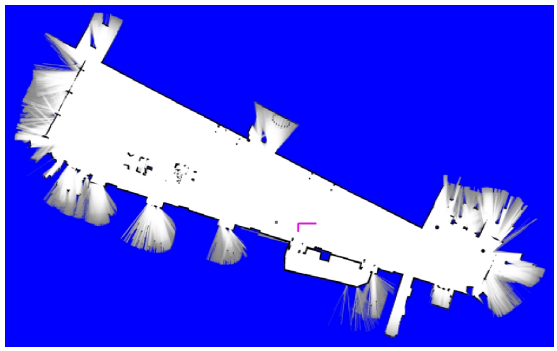

Fig 6. FR101 building map, computed in 0,52secs with GPU

Fig. 6 shows the computed map, and the results presented in Table I show the same gain as in the previous data set. As the complexity of the grid mapping is constant time O(1) for each measurement update it is logical that the computational savings proportion remains constant irrespective of the data set and map size.

Table II shows the importance of an appropriate use of device memory. It has been built using the laptop configuration, and shows the relative performance of the GPU implementation compared with the CPU one. It can be seen that if all the data is stored in the device (graphics card) in global memory, the performance of the GPU is worse than the one of the CPU up to a 50% of computation time increment. Nevertheless, moving a small part of the data to shared memory, and making it be computed by just one thread of the block, the computational savings become clearly visible. The use of constant memory to provide faster read access to common parameters allows further savings.

TABLE II EFFECT OF MEMORY USE

Device memory use	Processing time compared with the CPU
All data in global memory, each thread computing beam data	150%
Common block (beam) data in shared memory, computed only by one thread	65%
Input parameters in read-only constant memory	50%

It should be said here that the final results obtained with the CPUs and the GPUs are not completely identical. Although the resulting maps are visually undistinguishable, if their difference is computed and shown, several very small differences are visible. These differences can be due to the different floating point arithmetic carried out by the GPUs. However, these small differences are absolutely irrelevant for any navigation purpose.

7. CONCLUSION

This paper has presented a parallelization of the grid mapping algorithm as implemented in CARMEN, suitable to be run on a GPU. The GPU implementation has been proved to obtain large speedups, reducing the required computation time to 50% in the case of common graphics cards as those in mid end laptops, and down to 2% with high end (but affordable) graphics cards, compared with the same algorithm running on a CPU. Moreover it is important to highlight that the full computation is being targeted to the GPU, thus fully releasing the CPU, which can be used in parallel for other tasks.

Grid mapping falls into the category of mapping with known poses, so it requires other algorithm to correct the odometry drift and provide corrected poses. Many SLAM algorithms address this issue, and many of them are based on the idea of estimating the robot path as solution to the problem, and computing the resulting map afterwards. Those algorithms can greatly benefit from our approach. It can be seen from the experiments that the map resulting from a 200 meters trajectory can be computed in approximately 0,250 seconds, which in terms of mobile robots means practically real time, allowing the closure of large loops with an immediate computation of a complete global grid map.

It is a bit surprising that after the performance shown in Yguel et al (2008), the use of GPUs for processing in mobile robotics has not been widely adopted by the community. We believe that mobile robotics algorithms necessarily have to be adapted for heterogeneous (CPU and GPU) multi-core processors to benefit from modern hardware computational power and to be able to increase their performance up to the next level. Our next work will focus on the parallelization of a full SLAM algorithm, which will probably require the use of both the CPU and GPU, with the expectation of achieving and order of magnitude of speedup compared with a classical implementation.

ACKNOWLEDGMENT

This work was supported by Spanish Ministry of Science and Technology (Robonauta: DPI2007-66846-C02-01). Datasets have been obtained from the Robotics Data Set Repository Radish (2003), thanks to C. Stachniss

REFERENCES

Bresenham J. E.. (1965) Algorithm for computer control of a digital plotter, IBM Systems Journal, V 4, N.1, pp. 25-30

CARMEN, The Carnegie Mellon Robot Navigation Toolkit. Available at: http://carmen.sourceforge.net/

Elfes. A. (1987). Sonar-based real-world mapping and navigation. *IEEE Journal on Robotics and Automation*. Vol. 3 N. 3. pp. 249-265.

Halfhill T. R. Parallel Processing with CUDA. (2008). Microprocessor. NVIDIA url: http://www.nvidia.com

Moravec H., Elfes A. (1985) High resolution maps from wide angle sonar. IEEE International Conference on Robotics and Automation . Vol. 2. pp. 116-121. 1985.

Radish, The Robotics Data Set Repository. (2003) C. Stachhniss. URL: http://radish.sourceforge.net

Thrun S., Bücken A., Burgard W., Fox D., Fröhlinghaus T., Henning D., Hofmann T., Krell M., and Schmidt T.. (1998) Map learning and high-speed navigation in RHINO. AI-based Mobile Robots: Case Studies of Successful Robot Systems. MIT Press.

Thrun S., Leonard J. (2008). Simultaneous Localization and Mapping. Handbook of Robotics, chapter 37. Springer.. Siciliano, Bruno; Khatib, Oussama (Eds.) ISBN: 978-3-540-23957-4

Yguel M., Aycard O. and Laugier C..(2008) Efficient GPU-based Construction of Occupancy Grids Using several Laser Range-finders. International Journal of Vehicle Autonomous Systems. Issue: Volume 6, Number 1-2. Pages: 48 - 83