Filtro de partículas para localización de robots móviles implementado en arquitecturas multi-núcleo

Claudio J. Paz Gonzalo F. Perez Paina Luis R. Canali Julio H. Toloza
Centro de Investigación en Informática para la Ingeniería (CIII)
{cpaz, gperez, lcanali, jtoloza}@scdt.frc.utn.edu.ar

Resumen— **En este trabajo se presenta un análisis de desempeño de un algoritmo de localización de robots para operación en entornos estructurados, en diversas arquitecturas computacionales. El algoritmo bajo análisis se basa en un filtro de partículas, cuyo objetivo es estimar la ubicación del robot con respecto a un mapa global dado, comparando la lectura en tiempo real de un sensor LASER contra lecturas simuladas asociadas a un conjunto aleatorio de hipótesis de posiciones del robot. La comparación de desempeño se realiza en tres arquitecturas, incluyendo una implementación en procesadores gráficos (GPU).**

I. Introducción

En robótica móvil, una de las grandes ramas de investigación es la navegación autónoma de vehículos, tarea ésta que demanda que el robot conozca su posición y orientación con alta precisión. En entorno exteriores, el método usual para determinar la posición absoluta del vehículo es el GPS (*Global Positioning System*). Por otro lado, las aplicaciones usuales de robótica de servicio se desarrollan en ambientes techados como oficinas o depósitos. En estos entornos las señales de GPS son muy débiles, por lo tanto se debe recurrir a métodos de localización relativos. Los sensores odométricos tienen la desventaja de tener gran deriva debida al proceso de integración necesario para recuperar la posición.

La información odométrica puede ser de distintos orígenes, por ejemplo integración de desplazamientos de cámaras [1], unidades inerciales [2] o codificadores ópticos fijos a las ruedas del vehículo [3].

Un método que no tiene deriva es el que utiliza la correlación de mapas. En este método, las lecturas de los sensores se usan para crear un mapa local centrado en el robot el que luego es comparado con el mapa del ambiente de trabajo buscando coincidencias. Si bien este método no tiene deriva, requiere un gran poder computacional.

Para generar los mapas locales se pueden utilizar sensores de rango LASER tipo LIDAR (*Light Detection and Ranging*) que consiste en un conjunto de mediciones en coordenadas polares.

Un modo de encarar el problema de correlación de mapa es mediante el uso de múltiples procesadores, en los cuales distintos bloques del mapa del entorno se correlacionan con el mapa local [4][5]. De esta forma, la velocidad de resolución del problema de localización crece linealmente (en teoría) con la cantidad de procesadores.

En este trabajo se utiliza un enfoque alternativo, el cual, en lugar de correlacionar mapas locales contra el mapa de entorno, se generan aleatoriamente múltiples hipótesis de posiciones del robot, a partir de las cuales se simulan mediciones de rango usando el mapa global. Estas simulaciones se comparan con las lecturas de rango que efectivamente adquiere el robot, seleccionado aquellas de mayor semejanza. Estas hipótesis así elegidas son luego actualizadas conforme a la información odométrica.

El método aquí evaluado se llevó a cabo mediante la formulación de un filtro de partículas del tipo *bootstrap* [6] asociado al modelo cinemático de un vehículo que se mueve en un plano, en donde cada partícula representa una posible localización y orientación del robot. El peso probabilístico de cada partícula se asigna en base al modelo propuesto para las mediciones de rango. El objetivo del filtro es entonces estimar la posición y la orientación del móvil por medio de lecturas de un sensor de rango y el conocimiento del mapa global del entorno.

El algoritmo computacional se implementó en una placa de video con capacidad GPGPU (*General-Purpose Computing on Graphics Processing Units*), comparando su desempeño contra una implementación del mismo filtro realizada con las bibliotecas *OpenMP* [7] corriendo en un servidor dedicado con 16 núcleos, así como en un procesador Intel Dual Core.

En la sección II se mencionan los trabajos relacionados con el procesamiento paralelo en distintas arquitecturas. En la sección III presenta los fundamentos del filtrado bayesiano, el filtro de partículas y los modelos de movimiento y medición se describen en la sección IV. La implementación de éste filtro en arquitecturas CPU y GPU se describen en la sección V. Por último los resultados y conclusiones se presentan en las secciones VI y VII, respectivamente.

II. Trabajos relacionados

En la literatura de los últimos años se encuentran varios trabajos que comparan las distintas implementaciones de los filtros de partículas en arquitecturas que permiten paralelización [8]. Las arquitecturas más utilizadas para paralelizar algoritmos en la última década son las FPGA (*Field Programmable Gate Array*) [9][10], los *clusters* de computadoras [11], las computadoras con multi-núcleos [12] y últimamente las unidades de procesamiento gráfico (GPU) en las placas de video [13].

Los trabajos sobre GPUs, en general, son aplicaciones realizadas en lenguaje de programación C sobre CUDA

(*Compute Unified Device Architecture*) [14][15]. Esta metodología se hizo muy popular por lo simple de su implementación en comparación con su competidor *OpenCL* [16].

III. Estimación bayesiana

Los filtros bayesianos secuenciales son métodos recursivos que permiten estimar el estado de un sistema, cuya dinámica posee componentes aleatorios modeladas por funciones de densidad de probabilidad (*fdp*), en base al estado previo del sistema y a la información sensorial más reciente [17], la cual también se modela probabilisticamente mediante una *fdp*.

Típicamente el sistema se modela mediante un proceso estocástico de la forma

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \mathbf{w}_{k-1}) \tag{1}$$

donde $\mathbf{x}_k$ es el estado del sistema en el tiempo discreto k, la función de transición del sistema $\mathbf{f}_k$ depende del instante k, y $\mathbf{w}_{k-1}$ es un ruido blanco con media cero cuya *fdp* es conocida.

Las lecturas de los sensores $\mathbf{z}_k$ son funciones del estado $\mathbf{x}_k$ de la siguiente forma

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k, \mathbf{v}_k) \tag{2}$$

donde la función de medición $\mathbf{h}_k$ depende del instante k (por ejemplo en sistemas con sensores de múltiples tasas de muestreo) y $\mathbf{v}_k$ también es ruido blanco con media cero y *fdp* conocida.

La estimación de la *fdp* posterior $p(\mathbf{x}_k|\mathbf{z}_{1:k})$ se realiza en dos pasos: primero se predice el estado en el instante k a partir del estado estimado en el instante previo por medio de la ecuación Chapman-Kolmogorov

$$p(\mathbf{x}_k|\mathbf{z}_{1:k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{k-1}|\mathbf{z}_{1:k-1})d\mathbf{x}_{k-1}, \tag{3}$$

en donde la matriz de transición $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ se determina a partir de (1). El siguiente paso consiste en actualizar la *fdp* a priori (3) mediante la identidad de Bayes

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) = \frac{p(\mathbf{z}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{z}_{1:k-1})}{p(\mathbf{z}_k|\mathbf{z}_{1:k-1})}, \tag{4}$$

en donde la verosimilitud $p(\mathbf{z}_k|\mathbf{x}_k)$ se calcula a partir de (2). El factor de normalización en (4) está dado por

$$p(\mathbf{z}_k|\mathbf{z}_{1:k-1}) = \int p(\mathbf{z}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{z}_{1:k-1})d\mathbf{x}_k. \tag{5}$$

En general, la solución formal dada por (3) y (4) no tiene uso práctico, en particular debido a la imposibilidad de calcular la integral en (5). Sin embargo, esta solución puede ser aproximada mediante técnicas tipo Monte Carlo dando lugar a los denominados filtros de partículas. Una variante particular de estos filtros, adaptado al tratamiento del problema planteado, se describe en la Sección IV-C.

IV. Localización de robots con filtro de partículas

A. Modelo cinemático

El modelo usado en la etapa de predicción corresponde al modelo odométrico probabilístico de movimiento presentado en [4] que se describe brevemente a continuación. El modelo usa las lecturas de movimiento relativo obtenidos de la odometría del robot para realizar una transformación, la cual está comprendida por una primera rotación, una traslación y luego una segunda rotación

$$\mathbf{u} = [\delta_{rot1}, \delta_{trans}, \delta_{rot2}]^T.$$

Dadas las lecturas de odometría al tiempo discreto $k-1$,

$$\mathbf{x}_{k-1}^{odom} = \left[x_{k-1}^{odom}, y_{k-1}^{odom}, \theta_{k-1}^{odom}\right]^T$$

y al tiempo discreto k,

$$\mathbf{x}_{k}^{odom} = \left[x_{k}^{odom}, y_{k}^{odom}, \theta_{k}^{odom}\right]^T,$$

la acción de control está compuesta por

$$\begin{aligned}
\delta_{rot1} &= atan2(y_k^{odom} - y_{k-1}^{odom}, x_k^{odom} - x_{k-1}^{odom}) - \theta_{k-1}^{odom} \\
\delta_{trans} &= \sqrt{(x_{k-1}^{odom} - x_k^{odom})^2 + (y_{k-1}^{odom} - y_k^{odom})^2} \\
\delta_{rot2} &= \theta_k^{odom} - \theta_{k-1}^{odom} - \delta_{rot1}
\end{aligned}$$

Asumiendo que estas variables son afectadas por ruido gaussiano

$$\begin{aligned}
\delta_{rot1} &= \hat{\delta}_{rot1} + \varepsilon_{rot1}, \qquad \varepsilon_{rot1} \sim \mathcal{N}(0, \sigma_{rot1}) \\
\delta_{trans} &= \hat{\delta}_{trans} + \varepsilon_{trans}, \qquad \varepsilon_{trans} \sim \mathcal{N}(0, \sigma_{trans}) \\
\delta_{rot2} &= \hat{\delta}_{rot2} + \varepsilon_{trans}, \qquad \varepsilon_{rot2} \sim \mathcal{N}(0, \sigma_{rot2})
\end{aligned}$$

con

$$\begin{aligned}
\sigma_{rot1} &= \alpha_1|\delta_{rot1}| + \alpha_2|\delta_{trans}| \\
\sigma_{trans} &= \alpha_3|\delta_{trans}| + \alpha_4(|\delta_{rot1}| + |\delta_{rot2}|) \\
\sigma_{rot2} &= \alpha_1|\delta_{rot2}| + \alpha_2|\delta_{trans}|
\end{aligned} \tag{6}$$

donde $\alpha_i, i = 1, \ldots, 4$ son los parámetros de movimiento específicos del robot usado.

Por lo tanto, el estado del robot evoluciona de acuerdo a

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}) \tag{7}$$

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \delta_{trans}\cos(\theta_{k-1} + \delta_{rot1}) \\ \delta_{trans}\sin(\theta_{k-1} + \delta_{rot1}) \\ \delta_{rot1} + \delta_{rot2} \end{bmatrix} \tag{8}$$

En el caso del filtro de partículas, en cada paso, debe hacerse evolucionar cada partícula con este modelo basándose en la lectura odométrica del robot

$$\begin{aligned}
\hat{\delta}_{rot1} &= \delta_{rot1} + \mathcal{N}(0, \sigma_{rot1}) \\
\hat{\delta}_{trans} &= \delta_{trans} + \mathcal{N}(0, \sigma_{trans}) \\
\hat{\delta}_{rot2} &= \delta_{rot2} + \mathcal{N}(0, \sigma_{rot2})
\end{aligned}$$

donde $\hat{\delta}_{rot1}$, $\hat{\delta}_{trans}$ y $\hat{\delta}_{rot2}$ son las entradas de control que se usan para cada partícula y σ_{rot1}, σ_{trans} y σ_{rot2} las desviaciones estándar calculadas en (6).

Dada para cada partícula, el cálculo de la nueva posición de la misma es

$$\begin{aligned} x' &= x + \hat{\delta}_{trans} cos(\theta + \hat{\delta}_{rot1}) \\ y' &= y + \hat{\delta}_{trans} sin(\theta + \hat{\delta}_{rot1}) \\ \theta' &= \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{aligned} \tag{9}$$

Se puede ver en la Fig. 1 un ejemplo de un robot avanzando y la evolución de las partículas hasta ese momento si todas comenzaran en la misma posición que el robot, es decir, cada partícula es una hipótesis sobre la posición del robot dados el estado actual y el previo sólo aplicando el modelo de movimiento.

B. Modelo de medición

La ecuación de medición incluye las lecturas del escaner LASER y el mapa $\mathcal{M}$

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k, \mathcal{M}) \tag{10}$$

donde $\mathbf{h}(\cdot)$ es una función no lineal que relaciona una posición en el mapa con la lectura que indica el escaner.

En el caso de la simulación de las lecturas que corresponden a la posición de cada partícula, son generadas mediante un algoritmo trazado de rayos. En el algoritmo 1 se muestra el funcionamiento de este método, que consiste en hacer avanzar un punto en la dirección del LASER desde la ubicación de la partícula hasta que impacta con algún punto ocupado del mapa.

Entonces, (10) se puede expresar como la densidad de probabilidad condicional $p(\mathbf{z}_k|\mathbf{x}_k, \mathcal{M})$ que se puede aproximar como

$$p(\mathbf{z}_k|\mathbf{x}_k, \mathcal{M}) = \prod_{m=1}^{M} p(\mathbf{z}_k^m|\mathbf{x}_k, \mathcal{M}) \tag{11}$$

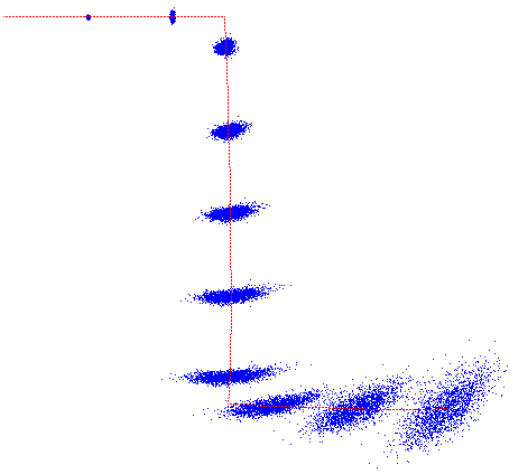

Fig. 1. Posibles posiciones del robot dado el modelo de movimiento

Algoritmo 1 Trazado de rayo

```
function ray_tracing(pos, ang, map)
    laser_pos ← pos;
    range ← 0;
    while range < LASER_MAX_RANGE do
        laser_pos ← forward(laser_pos, ang);
        range ← dist(pos, laser_pos);
        if map(laser_pos) is occupied then
            break;
        end if
    end while
    return range;
end function
```

donde

$$\mathbf{z}_k = \left[z_k^1, \ldots, z_k^M\right]^T \tag{12}$$

es el conjunto de M mediciones realizadas por el LASER correspondientes al tiempo discreto k.

C. Filtro de partículas

El filtro de partículas es una aproximación tipo Monte Carlo del filtro bayesiano secuencial descripto brevemente en la Sección III. Este filtro aproxima la *fdp* a posteriori por medio de un conjunto de muestras aleatorias con pesos asociados en función de esta distribución y estima el estado del robot en base a estos pesos [17]. Estas muestras son las llamadas partículas, y en éste modelo representan las distintas hipótesis de localización del robot.

La estimación del estado se realiza en un proceso iterativo que consta de las tres etapas: *predicción*, *actualización* y *remuestreo*.

En la etapa de predicción se utiliza el modelo de movimiento para hacer evolucionar cada partícula. De esta forma, en el tiempo discreto k la partícula $\mathbf{x}_k^i$ esta relacionada con la partícula $\mathbf{x}_{k-1}^i$ mediante (1) donde $\mathbf{u}_k$ es el vector de entrada del sistema.

La actualización consiste en evaluar cada partícula $\mathbf{x}_k^i$, con el modelo de medición para obtener $\mathbf{z}_k^i$, de esta forma se genera una medición correspondiente a ese estado hipotético que se compara con la medición obtenida a partir de los sensores $\mathbf{z}_k$ para obtener la verosimilitud de esa partícula. A mayor verosimilitud, mayor el peso q_k^i asociado a la partícula $\mathbf{x}_k^i$

$$q_k^i = q_{k-1}^i \, p(z_k|\mathbf{x}_k^i). \tag{13}$$

El remuestreo es necesario ya que el filtro de partículas adolece de un problema ampliamente estudiado llamado degeneración de las muestras, donde luego de pocas iteraciones una partícula tiene un gran peso y las demás tienen pesos muy próximos a cero. En [6] se propone una forma de solucionar este problema de degeneración la cual consiste en sustituir las partículas de menor peso por copias de aquellas de mayor peso.

V. Implementación de algoritmos en CPU y GPU

Para las pruebas del algoritmo, se implementó un simulador escrito en lenguaje de programación C++, en donde

se puede cargar un mapa del cual se extraerán las lecturas del escaner LASER del robot y las partículas en la etapa de actualización.

En el análisis de carga computacional, la etapa de mayor demanda de tiempo es la encargada de generar las lecturas de los sensores correspondientes a cada partícula con el método de trazado de rayos descripto en el algoritmo 1. La siguiente etapa de gran consumo computacional es la de remuestreo en el filtro de partículas.

El algoritmo para múltiples procesadores se implementó con las bibliotecas *OpenMP*. Este fue el único método de optimización para esta parte del código.

En el caso de la GPU, se utilizó la arquitectura CUDA para programar el filtro. Si bien se pasaron todas las funciones al código de CUDA, la implementación no es la óptima ya que en algunas partes del código, sólo se implementaron las optimizaciones básicas de CUDA y no las más avanzadas como el uso de memoria compartida, implementación por bloques, etc.

El código implementado consta de un kernel para cada etapa del filtro de partículas con el objetivo de facilitar el desarrollo como se puede ver en la Fig. 2. Se tuvo en cuenta que el mayor cuello de botella en los desarrollos sobre GPU son las transferencias de datos entre el CPU y la placa de video, de esta forma, aunque hay varios kernels y constantemente se devuelve el control al CPU, la transferencia de datos es mínima.

La etapa de predicción se muestra en el algoritmo 2 donde se observa que se utiliza el modelo de movimiento del robot, por lo que necesita que se le pase como parámetro la odometría del móvil para hacer evolucionar a cada partícula en base a este modelo.

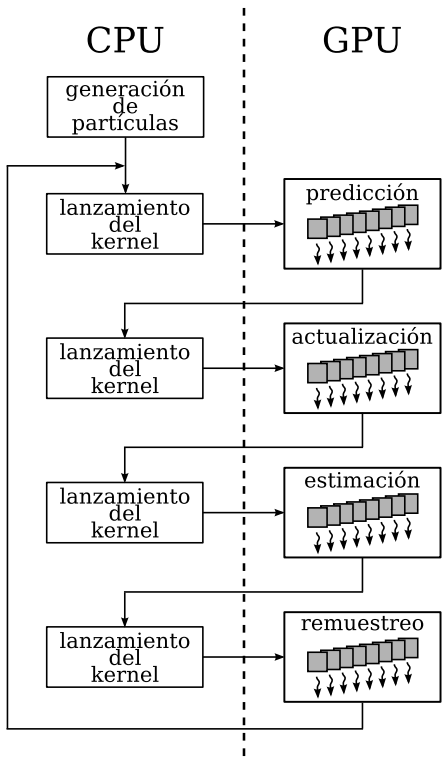

Fig. 2. Organización de las tareas entre el CPU y el GPU

Algoritmo 2 Predicción

function $process(particles, robot\_odom)$
  for $i = 1 \rightarrow N\_PARTICLES$ **do**
    $new\_particle(i) = \mathbf{f}(particle(i), robot\_odom, \mathbf{w})$;
  end for
  return $new\_particles$;
end function

Algoritmo 3 Actualización

function $update(sim\_laser\_scan, robot\_laser\_scan)$
  $\hat{\mathbf{z}} \leftarrow sim\_laser\_scan$;
  $\mathbf{z} \leftarrow robot\_laser\_scan$;
  $q \leftarrow (2\pi)^{-1/2} |\Sigma|^{-1/2} \exp(-\frac{1}{2}(\mathbf{z} - \hat{\mathbf{z}})^T \Sigma^{-1} (\mathbf{z} - \hat{\mathbf{z}}))$;
  return q;
end function

Algoritmo 4 Remuestreo

function $resample(particles, weights)$
  for $i = 1 \rightarrow N\_PARTICLES$ **do**
    $u \leftarrow \mathcal{N}(0, 1]$;
    $l \leftarrow 0$;
    while $u > weight(i)$ **do**
      $l++$;
    end while
    $new\_particle(i) \leftarrow particle(l)$;
  end for
  return $new\_particles$;
end function

El algoritmo 3 muestra como se puede implementar la actualización, con las lecturas simuladas de los LASER de cada partícula y las lecturas obtenidas por el robot.

El remuestreo es la etapa más importante ya que es aquí donde se seleccionan las partículas que tienen más verosimilitud para que continúen con el proceso. Las partículas que tienen menor peso son eliminadas, y copias de las mejores se ponen en su lugar. Puede verse en el algoritmo 4 que debido al bucle *while* dentro de la estructura repetitiva *for* esta etapa es la segunda de más carga computacional.

VI. Resultados

Para evaluar el rendimiento del filtro de partículas en CPU multi-núcleo, se ejecutó el algoritmo en un Servidor para rack PowerEdge R715 que cuenta con 16 núcleos AMD Opteron a una velocidad clock de 2GHz cada uno y una memoria RAM total de 16GB. Para probar el rendimiento del algoritmo en GPU, se utilizó una GPU NVIDIA GTX560 con 1GB de RAM y 336 CUDA Cores, montada en un Intel Dual Core de 2.13GHz y 2GB RAM. Como referencia se utilizó el Dual Core corriendo el algoritmo no paralelizado.

Los algoritmos se probaron con distintas configuraciones de partículas y cantidades de haces en el barrido del escaner LASER. Con estas configuraciones se midieron los errores en la localización y el tiempo que tomó cada filtro.

TABLA I
RENDIMIENTO DEL DUAL CORE CON 50, 100 Y 200 LECTURAS DE LASER Y 500, 1000, 2000, 3000 Y 4000 PARTÍCULAS

Partículas	Dual Core		
	50	100	200
500	**0.024116s**	0.047149s	0.09326s
1000	0.048838s	0.094874s	0.186954s
2000	0.100629s	0.192564s	0.377385s
3000	0.156249s	0.294173s	0.570679s
4000	0.21409s	0.398343s	0.767899s

TABLA II
RENDIMIENTO DE LA GPU CON 50, 100 Y 200 LECTURAS DE LASER Y 500, 1000, 2000, 3000 Y 4000 PARTÍCULAS

Partículas	GTX560		
	50	100	200
500	**0.0083717s**	**0.016783s**	**0.035316s**
1000	**0.016369s**	**0.032414s**	0.067133s
2000	**0.025286s**	0.049757s	0.10148s
3000	**0.033914s**	0.067074s	0.13055s
4000	0.042073s	0.083439s	0.16423s

Para calcular el tiempo que demanda cada configuración se uso el peor caso posible, donde la simulación de los sensores para cada partícula no detecta ningún obstáculo.

Como parámetro se considero que los sensores LIDAR comerciales tienen como tiempo de barrido $40ms$. De esta forma en las Tablas I, II y III se resaltan las configuraciones posibles de acuerdo a ésta restricción (las menores a $40ms$ son válidas).

Se puede ver que utilizando el Dual Core, la restricción sólo se cumple en la configuración de 500 partículas y 50 haces de medición.

Para el GPU las configuraciones posibles dentro de las restricciones son más numerosas, pero sin poder superar a la implementación del Opteron. No es sorpresivo que la GPU esté por debajo del servidor de 16 núcleos ya que la implementación utilizada para esta evaluación en el GPU es la más básica y no se implementaron optimizaciones.

Además, la implementación de un filtro de partículas en general, tiene el cuello de botella en cuanto a la velocidad de ejecución en la etapa de remuestreo, donde debe calcular la suma acumulada de las partículas, tarea que no es posible paralelizar. De todas maneras, el tiempo demandado por estas últimas dos arquitecturas es del mismo orden de magnitud (sólo 2 veces más rápido en el caso del Opteron).

TABLA III
RENDIMIENTO DEL OPTERON 16 NÚCLEOS CON 50, 100 Y 200 LECTURAS DE LASER Y 500, 1000, 2000, 3000 Y 4000 PARTÍCULAS

Partículas	Opteron 16 Núcleos		
	50	100	200
500	**0.0015640s**	**0.005621s**	**0.010434s**
1000	**0.0069301s**	**0.012090s**	**0.021029s**
2000	**0.015966s**	**0.025951s**	0.042716s
3000	**0.028121s**	0.042018s	0.069182s
4000	0.040781s	0.059155s	0.097497s

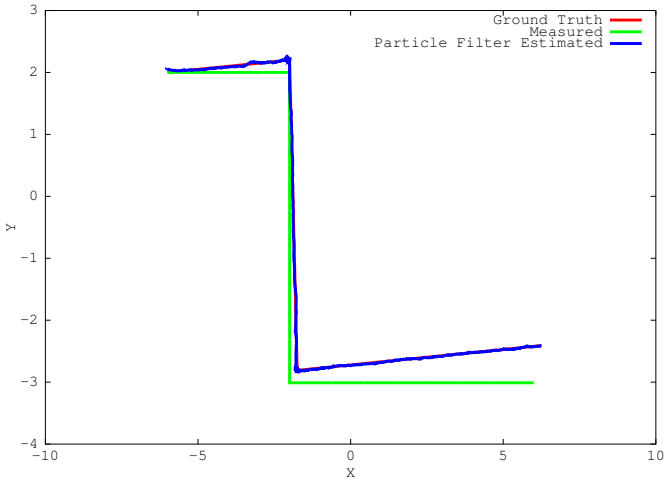

Fig. 3. Localización con deriva de odometría

Para comprobar el funcionamiento del filtro en los casos donde se superaban las restricciones, se evaluaron los errores en x, y y θ como se puede ver en (4). En este caso, se muestra la configuración 2000 partículas y 200 mediciones del LASER implementado en el Opteron de 16 núcleos. También se puede ver en la Fig. 3 que el filtro implementado tiene muy poco error aunque la odometría tenga mucha deriva.

Para todos los casos se evaluó la misma trayectoria de 17 metros. Todas las configuraciones mostraron errores por debajo de 5 centímetros salvo algunos picos de error de 10 centímetros en los momentos en que el robot se detenía para girar.

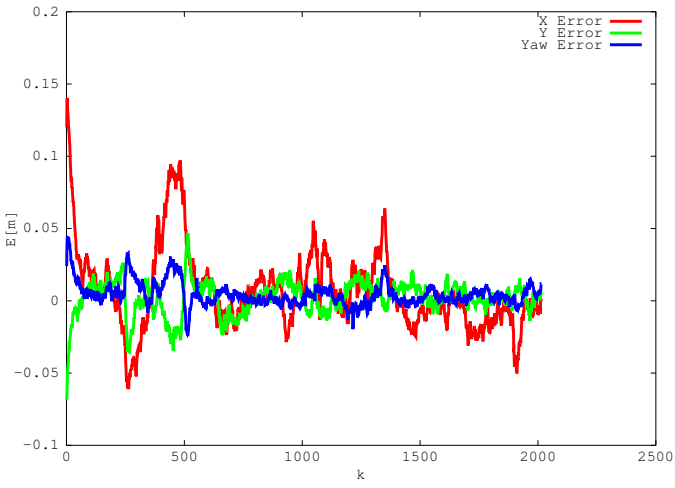

Fig. 4. Error de la estimación

VII. Conclusiones y trabajos futuros

Los ensayos realizados muestran que es posible resolver el problema de localización con una implementación corriendo en una GPU en ambientes estructurados como oficinas o depósitos.

Las pruebas muestran que incluso la implementación más básica del filtro corriendo en la GPU tiene buen rendimiento frente a la implementación corriendo en un procesador de escritorio. Si bien el servidor multi-núcleo es el más veloz, en este caso, es poco viable montar un rack en un robot móvil. Además, los costos de adquirir un servidor de estas características es de un orden de magnitud mayor que una placa de video como la usada.

El siguiente paso es optimizar el algoritmo de la GPU con las expectativas de superar a la implementación de OpenMP. También utilizar el mapa en forma vectorial, de forma que el tiempo de cálculo de la simulación del escaner LASER sea menor.

Agradecimientos

El primer autor se financia con el programa de becas de la Universidad Tecnológica Nacional. Este trabajo se enmarca dentro del proyecto "Guiado de Vehículos Autónomos usando Fusión de Señales de GPS de Bajo Costo y otros Sensores", PICT-PRH-2009-0136.

Referencias

[1] M. Fiala and A. Ufkes, "Visual odometry using 3-dimensional video input," *Canadian Conference on Computer and Robot Vision*, 2011.

[2] J. Shen, D. Tick, and N. Gans, "Localization through fusion of discrete and continuous epipolar geometry with wheel and imu odometry," *American Control Conference*, 2011.

[3] M. Malvezzi, P. Toni, B. Allotta, and V. Colla, "Train speed and position evaluation using wheel velocity measurements," *IEEE International Conference on Advanced Intelligent Mechatronics Proceedings*, 2001.

[4] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2005.

[5] A. U. Peker, O. Tosun, and T. Acarman, "Particle filter vehicle localization and map-matching using map topology," *IEEE Intelligent Vehicles Symposium*, 2011.

[6] N. Gordon, D. Salmond, and A. Smithdt, "Novel approach to nonlinear/non-gaussian bayesian state estimation," *IEEE Proceedings F, Radar and Signal Processing*, 1993.

[7] B. Chapman, G. Jost, and R. V. D. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[8] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *IEEE Transaction On Parallel And Distributed System*, 2011.

[9] M. Lifeng, J. Zhang, C. Chakrabarti, and A. Papandreou-Suppappola, "A new parallel implementation for particle filters and its application to adaptive waveform design," *IEEE Workshop on Signal Processing Systems*, 2010.

[10] I. Zuriarrain, J. I. Aizpurua, F. Lerasle, and N. Arana, "Multiple-person tracking devoted to distributed multi smart camera network," *IEEE International Conference on Intelligent Robots and Systems*, 2010.

[11] X. Yan, W. Zhang, H. Deng, and S. Bu, "The parallelization of three-dimensinal electro-magnetic particle model using both mpi and openmp," *IEEE International Conference on Computational and information Sciences*, 2010.

[12] T. Dinh, Q. Yu, and G. Medioni, "Real time tracking using an active pan-tilt-zoom network camera," *IEEE International Conference on Intelligent Robots and Systems*, 2009.

[13] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm," *IEEE International Conference on Parallel and Distributed Computing, Application and Technologies*, 2011.

[14] K. Par and O. Tosun, "Parallelization of particle filter based localization and map matching algorithm on multicore/manycore architectures," *IEEE Intelligent Vehicles Symposium*, 2011.

[15] M.-A. Chao, C.-Y. Chu, C.-H. Chao, and A.-Y. Wu, "Efficient parallelizad particle filter design on cuda," *IEEE Workshop on Signal Processing Systems*, 2011.

[16] R. Gaetano and B. Pesquet-Popescu, "Opencl implementation of motion estimation for cloud video processing," *IEEE International Workshop on Multimedia Signal Processing*, 2011.

[17] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, 2002.