Universidad Carlos III de Madrid

Departamento de Ingeniería de Sistemas y Automática

PROYECTO FIN DE CARRERA

Aceleración de algoritmos de localización para robots móviles en 3D mediante CUDA®

Autor: Jorge Luis Lovaco Hernández
Tutor: Fernando Martín Monar
Leganés, noviembre de 2014

No pude despedirme y tampoco pude decírtelo.

Gracias papá.

Agradecimientos

Durante años una serie de pérdidas irreparables me han hicieron sentirme perdido, frustrado y sólo. Inesperadamente fue esta soledad la que iluminó mi camino. En las horas que pasé evadiéndome de mí mismo encontré un día la documentación que NVIDIA™ publicaba en su página web sobre la tecnología CUDA®. La curiosidad me llevó a investigar sobre el tema con pasión, y la casualidad me llevó a encontrar el presente proyecto ofrecido por mi tutor Fernando. Comenzarlo fue el principio del fin, por fin he terminado mis estudios y he conseguido salir del pozo en el que estaba metido. Por ello mi primer agradecimiento va dedicado a él, por haberme dado la oportunidad de enfrentarme al desafío que la programación en paralelo supone, por su paciencia conmigo y por estar siempre disponible para las dudas que surgían.

Los siguientes que tengo que mencionar son mi madre y mi hermano. Soy una persona muy difícil de tratar y ellos nunca me han dejado. Gracias mamá por soportarme. Gracias Javi por tus momentos de apoyo y guía, aunque para ello tengas que chincharme de vez en cuando.

En estos años he tenido muchos compañeros, pero sólo recuerdo a cuatro. A Fer y Luismi, que me ayudaron en el comienzo. A Jano con el que disfruté hablando de música y guitarras. Y por último, pero no por ello el que menos, a ti Rober, que apareciste por sorpresa y sorprendentemente nunca has desaparecido.

Muchas gracias también por los buenos ratos a Miguel, Raúl, Mateo, Alvarito, Albert, Nacho, Román, Lydia, Zaida, Patri. Gracias a mis chicas Silvia, Raquel y Laura. No tengo palabras para vosotras, aunque afortunadamente existe una que resume todo. Gracias.

A mis “herramientados” Guille, Jare, Miguelito y Jessica. Por esos ratos inolvidables que nunca podremos contar y por ser capaces de soportarme incondicionalmente. También a Alberto y Lara, por los coloquios interminables, y a Jóse y Lydia por sus consejos sobre C.

Gracias a los colegas de toda la vida, Miguel, Andrés, Luixi, LLopis, Joaquín, Cardo y Jimmy. Nunca se vio tanta locura en unas torres. Aquí debo mencionar por partida doble a Jimmy, gracias por tu compañía y por todo lo que compartiste aquél verano de 2004. Jamás te lo agradeceré bastante y por desgracia ya no podré hacerlo. Nunca te olvidaré.

Debo mencionar a la gente del servicio de informática tanto de Getafe como Leganés, por ser como una segunda familia y por todo lo que me han enseñado. Entre ellos especialmente a la gente de la quiniela, Paquito, Sergio, Javi, Fernan, Raúl, Eulogio y Mikel. No hemos visto un céntimo, pero tampoco ha hecho falta. Mención especial para Rodri, tu apoyo, ayuda, ánimo y amistad son incomparables.

También quiero mencionar a Peter y a toda su “vanguardia”, que no creo que llegue a saber lo que consiguió ni cuánto se lo agradezco. A Marinella por sus consejos, molte grazie. A Soraya, por literalmente salvarme la vida y conseguir hacerme arrancar.

Termino recordando a mi tía Conchi, a mi abuelo Gabriel y a mi padre. Gracias por todo lo que hicisteis. No os habéis ido, sé que seguís conmigo.

Resumen

La localización de un robot es uno de los problemas a los que se enfrenta actualmente la robótica moderna. El algoritmo desarrollado en la Universidad Carlos III, llamado RELF-3D, intenta resolver este problema, centrándose en hacerlo cuando no hay conocimiento acerca de la posición de partida. El robot MANFRED-V2 está programado con él para encontrar su posición haciendo comparaciones con un mapa precargado de uno de los edificios del campus. Mide las distancias a las diferentes paredes que hay en su posición actual con varios escáneres láser, a continuación, se ejecuta el algoritmo RELF-3D. Este algoritmo se basa en la adaptación genética, creando poblaciones aleatorias que representan posiciones dentro del mapa y haciendo que evolucionen a una solución óptima.

Sin embargo, el algoritmo necesita hacer varios cálculos para calcular las distancias, evaluar sus errores y por lo tanto se requiere una gran cantidad de tiempo.

Este proyecto ha sido propuesto para encontrar una manera de resolver este problema. El objetivo es llegar a un tiempo de cálculo aceptable utilizando la computación paralela. Por lo tanto un lenguaje de programación debe ser elegido. La decisión fue utilizar las bibliotecas CUDA® de NVIDIA ™ para el lenguaje C, debido a su proximidad con el lenguaje C original y la disponibilidad de las arquitecturas CUDA® en la casi totalidad de las GPUs de esta compañía por un precio razonable.

El resultado obtenido superará expectativas del lector debido a la cantidad de tiempo ahorrado mediante el uso de suficientes hilos paralelos para resolver todas las posibles distancias medidas al mismo tiempo.

Varias propuestas para desarrollos futuros del algoritmo RELF-3D se hacen después de las conclusiones, así como algunas proposiciones de mejora del hardware.

Abstract

Locating a robot is one of the main problems faced nowadays by modern robotics. The algorithm developed in the Carlos III University called RELF-3D tries to solve this problem, focusing on doing it when there is no knowledge about the starting position. The robot MANFRED-V2 is programmed with it to find its position by making comparisons with a preloaded map of one of the buildings of the campus. The laser scan is used to measure distances to the walls, and then the algorithm RELF-3D is executed. This algorithm is based on adaptive genetics, creating randomized populations that represent positions within the map and evolving them to an optimal solution.

However, the algorithm needs to make several computations to calculate the distances, evaluate its errors and hence a lot of time is required.

This project has been proposed to find out a way to solve this problem. The goal is to reach an acceptable computational time using parallel computing. Therefore a programming language must be chosen. The decision was to use the CUDA® libraries from NVIDIA™ for the C language, because of its proximity to the original C language and the availability of the CUDA® architectures in almost all of this company GPUs for a reasonable price.

The result obtained will surpass the reader's expectative because of the amount of time saved by using enough parallel threads to solve all the possible distances measured at the same time.

A few proposals for future developments of the RELF-3D algorithm are made after the conclusions, as well as some hardware improvements.

Índice General

Índice de figuras………-11-

Índice de tablas………-13-

CAPÍTULO 1: INTRODUCCIÓN, ESTADO DEL ARTE …………………………………………………………….-14-

1.1 Motivación del proyecto: Complejidad y coste computacional………………………………….-17-

1.2 Trabajo relacionado……………………………………………………………………………………………….-18-

1.2.1 Algoritmos genéticos de interés histórico…………………………………………………………-19-

1.3 Estructura de la memoria………………………………………………………………………………….-21-

CAPÍTULO 2: FUNDAMENTOS TEÓRICOS.………………………………………………………………………-22-

2.1 Formulación y solución del problema de localización…………………………………………..-22-

2.2 Filtro de localización evolutiva……………………………………………………………………………..-22-

2.2.1 Mutación cruce y selección………………………………………………………………………….-23-

2.2.2 Mecanismo de thresholding…………………………………………………………………………..-23-

2.2.3 Mecanismo de descarte………………………………………………………………………………..-24-

2.2.4 Condiciones de convergencia…………………………………………………………………………-24-

2.2.5 Función de aptitud………………………………………………………………………………………..-25-

CAPÍTULO 3: FUNDAMENTOS DE CUDA™ (COMPUTE UNIFIED DEVICE ARCHITECTURE)…….-27-

3.1 Desarrollo histórico de las unidades de proceso……………………………………………………..-27-

3.2 El auge del cálculo a través de las unidades de proceso gráfico…………………………….-27-

3.2.1 Breve historia de las GPUs……………………………………………………………………………..-27-

3.2.2 Computación GPU temprana………………………………………………………………………….-28-

3.3 Alternativa al modelo secuencial: CUDA™……………………………………………………………..-29-

3.4 Arquitectura de una GPU moderna………………………………………………………………………..-30-

3.4.1 Arquitectura NVIDIA® Kepler…………………………………………………………………………..-31-

3.5 Relación velocidad y cálculo paralelo……………………………………………………………………-32-

3.5.1 Aplicaciones……………………………………………………………………………………………….-33-

3.5.1.1 Imágenes médicas……………………………………………………………………………..-33-

3.5.1.2 Dinámica de fluidos…………………………………………………………………………….-33-

3.5.1.3 Ciencia medioambiental………………………………………………………………….-34-

CAPÍTULO 4: ALGORITMO RELF-3D ACELERADO……………………………………………………….-35-

4.1 Conceptos básicos de programación CUDA™……………………………………………………………-35-

4.1.1 Las 3 reglas de la programación de GPGPUs…………………………………………….-36-

4.1.2Conceptos algebraicos……………………………………………………………………………….-36-

4.1.3 Configuración de la GPU………………………………………………………………………….-38-

4.1.3.1 Creación del kernel……………………………………………………………………….-38-

4.1.3.2 Gestión de memoria……………………………………………………………………..-40-

4.1.3.3 Gestión de errores……………………………………………………………………….-42-

4.1.3.4 Liberación de recursos………………………………………………………………….-44-

4.2 Descripción del programa…………………………………………………………………………………….-44-

4.2.1 Creación de un entorno tridimensional nuevo…………………………………………-44-

4.2.2 Creación del kernel……………………………………………………………………………………..-47-

4.2.3 Adaptación del algoritmo genético………………………………………………………….-53-

CAPÍTULO 5: RESULTADOS EXPERIMENTALES………………………………………………………………-54-

5.1 Resultados del entorno tridimensional vacío……………………………………………………..-55-

5.2 Resultados del entorno tridimensional con obstáculos…………………………………………..-59-

5.3 Resultados medios totales…………………………………………………………………………………….-64-

5.4 Experimentos alternativos…………………………………………………………………………………..-64-

5.5 Comentarios acerca de los resultados……………………………………………………………………..-65-

CAPÍTULO 6: CONCLUSIONES Y TRABAJO FUTUROS…………………………………………………….-66-

6.1 Consecución de objetivos……………………………………………………………………………………..-66-

6.2 Ventajas de desarrollo en CUDA™……………………………………………………………………………..-66-

6.3 Desarrollo portable…………………………………………………………………………………………………-66-

APÉNDICE A: COSTES DEL PROYECTO…………………………………………………………………………..-68-

APÉNDICE B: MANUAL DEL USUARIO………………………………………………………………………….…..-69-

APÉNDICE C: CÓDIGO DEL PROYECTO…………………………………………………………………………….-75-

REFERENCIAS BIBLIOGRÁFICAS…………………………….………………………………………………………..-114-

Índice de figuras

Figura 1.1: Robot Manfred-V2………………………………………………………………………………………….-15-

Figura 1.2: Telémetro láser SICK-PLS …………………………………………………………………………..…-16-

Figura 1.3: Entorno visual original y su lectura por láser……………………………………………..………-16-

Figura 1.4: Mapa virtual……………………………………………………………………………………………………….-18-

Figura 3.1: Ejemplo de juego con entorno 3D. "Duke Nukem 3D"…………….. http://www.3drealms.com...-28-

Figura 3.2: Diferencias de construcción. "Programming Massively Parallel Processors. A Hands-on Approach." Kirk, David B., Hwu, Wen-mei W……………………………………………………………..-30-

Figura 3.3: Chip de NVIDIA®. www.nvidia.es..-31-

Figura 3.4: Arquitectura CUDA. "NVIDIA GeForce GTX 680 Whitepaper. The fastest, most efficient GPU ever built." http://www.nvidia.es..-32-

Figura 3.5: Comparativa de operaciones de coma flotante. "Programming Massively Parallel Processors. A Hands-on Approach." Kirk, David B., Hwu, Wen-mei W……………………….………-33-

Figura 4.1: Representación Grid…………………………………………………………………………………………….-36-

Figura 4.2: Conversión matriz bidimensional en direccionamiento lineal………………….……….-37-

Figura 4.3: Producto matricial………………………………………………………………………………………………..-37-

Figura 4.4: Asignación de memoria………………………………………………………………………………………..-41-

Figura 4.5: Representación entorno tridimensional……………………………………………………………-45-

Figura 4.6: Representación entorno tridimensional…………………………………………………………..-45-

Figura 4.7: Representación nube de puntos masiva…………………………………………………………-46-

Figura 4.8: Habitación con vista simple……………………………………………………………………………...-46-

Figura 4.9: Habitación con paredes……………………………………………………………………………………….-47-

Figura 4.10: Giro horizontal…………………………………………………………………………………………………..-48-

Figura 4.11: Inclinación………………………………………………………………………………………………….…-49-

Figura 4.12: Esquema coordenadas esféricas. Schmidtke, Romero……………………………………..-49-

Figura 4.13: Giro horizontal…………………………………………………………………………………………………-50-

Figura 4.14: Ángulo de inclinación……………………………………………………………………………..-50-

Figura 4.15: Inclinación……………………………………………………………………………………………….-50-

Figura 4.16: Proyección vector……………………………………………………………………………………..-51-

Figura 5.1: Posición (3, 3, 1, 0)……………………………………………………………………………….......-55-

Figura 5.2: Posición (10, 10, 1, 0)…………………………………………………………………………….....-56-

Figura 5.3: Posición (15, 15, 1, 0)………………………………………………………………………………-57-

Figura 5.4: Posición (22, 5, 1, 0)…………………………………………………………………………………-58-

Figura 5.5: Posición (2, 2, 1, 0)………………………………………………………………………………..….-59-

Figura 5.6: Posición (2, 13, 1, 0)……………………………………………………………………………….-60-

Figura 5.7: Posición (2, 18, 1, 0)……………………………………………………………………………...-61-

Figura 5.8: Posición (15, 2, 1, 0)………………………………………………………………………………..-62-

Figura 5.9: Posición (15, 15, 1, 0)………………………………………………………………………….…-63-

Figura 5.10: Posición (3, 10, 1, 0)……………………………………………………………………………...-64-

Figura 6.1: NVIDIA® Jetson TK1. www.nvidia.es...-67-

Figura B.1: Crear nuevo proyecto………………………………………………………………………………-69-

Figura B.2: Opción Win32………………………………………………………………………………………....-70-

Figura B.3: Nombrar…………………………………………………………………………………………………..-70-

Figura B.4: Opciones de creación de proyecto……………………………………………………………-71-

Figura B.5: Opciones de personalización……………………………………………………………………..71-

Figura B.6: Personalización para CUDA™…………………………………………………………………….-71-

Figura B.7: Propiedades…………………………………………………………………………………………………-72-

Figura B.8: Incluimos los directorios…………………………………………………………………………….-72-

Figura B.9: Incluimos las librerías……………………………………………………………………………….-73-

Figura B.10: Librerías adicionales en menú Linker……………………………………………………..-73-

Figura B.11: Dependencias adicionales………………………………………………………………………..-74-

Figura B.12: Nombrar con extensión CUDA™……………………………………………………………..-74-

Índice de tablas.

Tabla 1.1 Tiempo de cálculo para un número fijo de iteraciones……………………………………..-17-

Tabla 1.2 Tiempo de cálculo para un tamaño de población fijado en 50 elementos…………..-17-

Tabla 5.1 Resultados posición. (3, 3, 1, 0)……………………………………………………………….…-55-

Tabla 5.2 Resultados posición. (10, 10, 1, 0)…………………………………………………………...-56-

Tabla 5.3 Resultados posición. (15, 15, 1, 0)…………………………………………………………….-57-

Tabla 5.4 Resultados posición. (22, 5, 1, 0)……………………………………………………………..-58-

Tabla 5.5 Resultados posición. (2, 2, 1, 0)……………………………………………………………….-59-

Tabla 5.6 Resultados posición. (2, 13, 1, 0)……………………………………………………………..-60-

Tabla 5.7 Resultados posición. (2, 18, 1, 0)…………………………………………………………….-61-

Tabla 5.8 Resultados posición. (15, 2, 1, 0)…………………………………………………………….-62-

Tabla 5.9 Resultados posición. (15, 15, 1, 0)…………………………………………………………...-63-

Tabla 5.10 Resultados posición. (3, 10, 1, 0)……………………………………………………………-64-

Tabla A.1 Inversión temporal………………………………………………………………………………..-68-

Tabla A.2 Coste total asociado al proyecto……………………………………………………………..-68-

Capítulo 1: Introducción, estado del arte

Las tareas cognitivas, como pueden ser la percepción de objetos y sus relaciones con el entorno, o la comprensión del lenguaje dentro de un contexto, o la planificación de acciones, todavía están lejos de las capacidades humanas para la inteligencia artificial pese a tener una precisión y rapidez de cálculo mayor que la humana.

¿Cuál es la base para esta diferencia? Una respuesta, quizás la que clásicamente se podría esperar de la inteligencia artificial, es "software". Si tuviéramos las premisas para desarrollar el programa adecuado, seríamos capaces de obtener la fluidez y adaptabilidad humanas.

Podría decirse que esta respuesta es parcialmente correcta. Ha habido grandes avances en nuestra comprensión de la cognición dando como resultado el desarrollo de lenguajes de programación de alto nivel y potentes algoritmos. Es indudable que estos avances continuarán, pero no podemos ceñirnos a ellos y pensar que ahí termina desafío.

Podemos considerar que las personas son más inteligentes que los ordenadores actuales debido a que el cerebro emplea una arquitectura "computacional" que está más adecuada al aspecto del procesamiento de tareas cognitivas. Estas tareas requieren, generalmente, la consideración simultánea de mucha información con posibles obstáculos. Cada obstáculo puede estar especificado de manera imperfecta y ambigua y, sin embargo, resultar decisivo en la determinación del procesado de la respuesta.

Esto nos lleva al problema de localización en robótica móvil. En gran cantidad de tareas un robot autónomo necesita saber su localización para poder realizarlas con éxito. Este problema es crucial en robótica y puede ser definido como la búsqueda de las coordenadas del robot relativas a su entorno, asumiendo que éste está proporcionado por un mapa. Un ejemplo de esto sería la ejecución de una ruta de manera exitosa durante la navegación. Esto nos lleva a distinguir entre dos situaciones diferentes:

-El problema de relocalización (el robot conoce su localización, al menos de manera aproximada).

-El problema de localización global (no hay conocimiento de la situación inicial). Este ha sido el objeto de estudio y para el que se desarrolló el método de localización "RELF-3D".

El método RELF-3D ("Rejection Evolutionary Localization Filter in Three Dimensions") es un algoritmo de localización sobre el que se implementarán y adaptarán las mejoras que se estudiarán en el presente proyecto. Se basa en la representación de la localización del robot por un conjunto de posibles estimaciones ponderado por una función de aptitud. El estado se estima de forma recursiva utilizando un conjunto de resultados seleccionados de acuerdo con el peso asociado a cada solución posible incluido en el conjunto. El conjunto de soluciones evoluciona en el tiempo para integrar la información de los sensores y la información de movimiento del robot.

Se ha trabajado sobre mapas 3D, siendo necesario el cálculo de cuatro coordenadas, correspondiendo éstas a la posición (x, y, z) y la orientación horizontal (θ). Los grados de libertad establecidos son 4 por simplicidad (ya que nuestro robot trabaja en 4 G.D.L), pero el robot podría trabajar en 6 G.D.L $(x, y, z, \psi, \theta, \phi)$. El método actual puede ser fácilmente expandido a 6 G.D.L. sin incrementar el coste computacional.

El motor de adaptación del método RELF-3D está basado en un mecanismo de adaptación evolutiva, el cual combina una búsqueda de gradiente estocástico junto con una búsqueda probabilística para encontrar los candidatos de posición más prometedores.

Figura 1.1: Robot Manfred-V2.

El grupo de investigación está trabajando en el desarrollo de la plataforma experimental Manfred-V2 (figura 1.1). Este robot tiene sensores incorporados que han sido usados para el desarrollo del algoritmo RELF-3D. El primero es un telémetro láser (SICK PLS - figura 1.2), el cual proporciona información en 3D sobre el entorno. Las medidas del sensor original son bidimensionales, pero se añadió un motor que le permite rotar arriba y abajo, permitiéndole obtener medidas en las tres dimensiones.

Figura 1.2: Telémetro láser SICK-PLS.

La lectura 3D del láser es usada por el módulo de localización para intentar situar el robot en un entorno familiar. Un entorno típico y una lectura 3D real del láser pueden verse en la figura 1.3. También tiene implementado un láser simulado, que trabaja en 13 exploraciones verticales separados 5 grados; cada exploración vertical contiene 61 lecturas horizontales separadas por 3 grados. Se debe hacer notar que, para el desarrollo del proyecto fin de carrera que nos atañe, se aumentó el número de lecturas horizontales hasta 181, de manera que pudiera hacerse más notable la computación en paralelo. Se entrará en detalle al respecto en sucesivos capítulos.

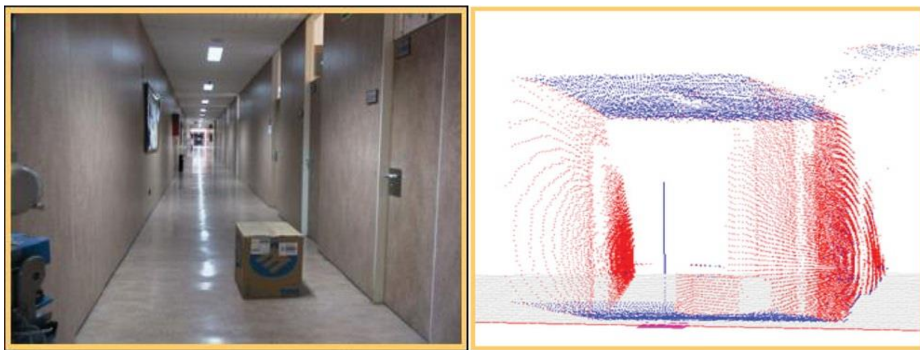

Figura 1.3: Entorno visual original y su lectura por láser.

Es necesaria, sin embargo, más información para resolver por completo el problema. Cuando el robot se mueve (el problema de localización global está resuelto, pero el problema de relocalización comienza), necesitamos entonces información sobre su movimiento. Con el fin de conseguirlo, el robot está equipado también con un codificador óptico (HEDS-5540), un sensor necesario para obtener la información odométrica. Esta información nos da una idea sobre el movimiento pero tiene diversas contrapartidas: la precisión es muy baja y el error se acumula con el tiempo.

1.1 Motivación del proyecto: Complejidad y coste computacional

Un último problema abordado en los experimentos se refiere al tiempo de ejecución del algoritmo RELF-3D. El coste computacional de cualquier algoritmo que trabaja en mapas 3D es muy alto, y este es uno de los mayores problemas encontrados por los investigadores en este campo. El tiempo depende de varios factores: la plataforma informática, el modelo de predicción de la observación y las mediciones del sensor, junto con el tamaño de la población y el número de iteraciones. La complejidad del algoritmo es O(n), lo cual significa que crece linealmente con el tamaño de la población.

Tiempo de cálculo para un número fijado de iteraciones.	
Población/iteraciones	Tiempo (s)
200/15	16.14
100/15	8.77
50/15	4.09
25/15	2.25
10/15	1.04

Tabla 1.1

Tiempo de cálculo para un tamaño de población fijado en 50 elementos.	
Población/iteraciones	Tiempo (s)
50/5	1.42
50/10	2.95
50/15	4.03
50/20	5.66
50/25	7.21
50/50	12.75

Tabla 1.2

Analizando el método de localización descrito, el cuello de botella desde un punto de vista computacional es la función llamada “dist_est_3d”. Esta consiste en la estimación de las mediciones tridimensionales por láser, compuestas de 9x61 rayos láser para una posición estimada en un entorno simulado, y debe ser calculado para cada miembro de la población. La lectura es usada por la función de aptitud para calcular el error.

Un punto crítico en cualquier algoritmo de localización global es la variación de los requisitos computacionales con las dimensiones del entorno. Los requisitos del algoritmo RELF-3D en los experimentos no dependen únicamente del tamaño del entorno, sino también de las características del lugar en el que el robot es localizado. Como por ejemplo, en los experimentos hechos en un entorno de $900m^2 \cdot 3m$ (figura 1.4), el algoritmo requiere al menos 50 muestras cuando está en el pasillo y 100 cuando está en uno de los despachos.

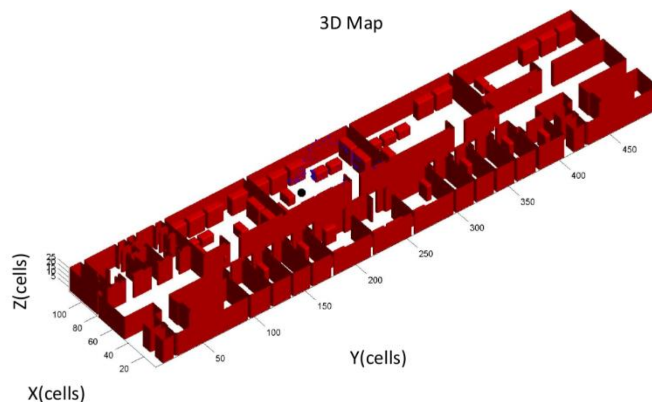

Figura 1.4: Mapa virtual.

Son la complejidad computacional y los tiempos de cálculo mostrados en las tablas 1.1 y 1.2 los que han motivado el presente proyecto. En él se intentará hacer uso de las nuevas tecnologías para estudiar la adaptación del RELF-3D a técnicas de cálculo en paralelo y la rentabilidad de las mismas en términos de tiempo y precisión. Para este estudio utilizaremos la plataforma de desarrollo proporcionada gratuitamente por NVIDIA™ basada en su arquitectura CUDA®. Gracias a esta tecnología se adaptará la parte más pesada, computacionalmente hablando, del método RELF-3D a una función que realizará los cálculos de manera paralela manteniendo intacta la parte de adaptación evolutiva. Para ello se estudiará la capacidad de transferencia de datos entre la unidad de proceso gráfico (GPU) que realizará los cálculos en paralelo; la estructura de hilos paralelos con la que se configurará la función, procurando que sea lo más genérica posible de cara a su portabilidad en futuros desarrollos; la configuración geométrica y algebraica óptima de cara a una fácil interpretación del usuario para futuras modificaciones; y por último, se desarrollaran dos entornos tridimensionales ficticios de cara a realizar las comparativas entre la programación secuencial y la programación paralela por medio de la comparativa de tiempos y la “ley de Amdahl”.

1.2 Trabajo relacionado

Hay diferentes familias de algoritmos que pueden proporcionar una solución al problema de la localización global en dos dimensiones: Métodos Bayesianos (filtros probabilísticos basados en cuadrículas y métodos de localización de Monte Carlo se pueden incluir aquí), métodos basados en optimización (filtros de evolución diferencial-DE y filtros de optimización de enjambres de partículas), y métodos híbridos (filtros multi-hipótesis de Kalman). Por ejemplo, *Burgard et al.*[2] ha estudiado el problema asumiendo que el robot está en un mapa de cuadrículas y *Fox et al.*[6] ha desarrollado un módulo de localización de Markov en entornos dinámicos. Este trabajo puede ser incluido en el campo de los basados en optimización. *Vahdat et al.*[26] aplica dos métodos evolutivos (optimización DE y de enjambre de partículas) y ellos comparan ambos con los métodos de localización Monte Carlo.

La mayoría de estos métodos han sido aplicados a entornos bidimensionales. Si hablamos del problema de la localización en un mapa tridimensional, es más difícil encontrar información al respecto. Por ejemplo, *Kümmerle et al.*[9] ha trabajado con Monte Carlo en aplicaciones al aire libre, aplicando un filtro de partículas para estimar completamente el estado hexadimensional del robot. Estos autores solventan el problema de la localización en entornos al aire libre haciendo coincidir las mediciones de rayos láser con un mapa previo del entorno usando mapeado de superficie multi-nivel (MLS). *Lingemann et al.*[11] ha desarrollado el algoritmo llamado "High-speed And Yet Accurate Indoor" (HAYAI) que localiza el robot haciendo coincidir los rasgos entre el conjunto de datos y el conjunto modelo, su contribución radica en un rápido filtrado y extracción de los rasgos naturales en las mediciones láser y una solución de forma cerrada para calcular el cambio de postura. *Tsai et al.*[26] fusiona sensores inerciales y ultrasónicos, y aplica un algoritmo basado en un filtro extendido de Kalman (EKF) para estimar la posición actual de un robot móvil navegando sobre un terreno irregular cubierto. *Lai et al.*[26] ha diseñado un método no lineal para un sistema compuesto de un telémetro láser y cuatro reflectores artificiales.

Existen también muchos grupos de investigación que usan técnicas de visión por computador. *Se et al.*[23] considera el problema de la localización global como un problema de reconocimiento de lugar, y lo soluciona haciendo coincidir las características de transformaciones de rasgos de escala invariante (SIFT), detectados en el marco actual, para pre construir la base de datos del mapa SIFT. *Ho* demuestra cómo el robot puede realizar la localización global usando un espejo panorámico en conjunción con un extensivo modelo tridimensional del entorno y un filtro de partículas para localización. *Royer et al.*[21] también usa un mapa construido previamente para calcular la localización del robot en tiempo real usando algoritmos de visión.

Existen adicionalmente numerosos grupos de investigación que no están exclusivamente centrados en solucionar el problema de la localización del robot, si no que también están interesados en el mapeado, considerando que el robot está en entornos tridimensionales. *Nütcher et al.*[14] ha desarrollado un método de mapeado robótico basado en analizar por láser del rango tridimensional de manera localmente consistente usando coincidencia iterativa de exploración del punto más cercano. *Triebel et al.*[25] trabaja con mapas tridimensionales creando mapas de superficie multinivel. *Cole* usa un láser buscador de alcance capaz de obtener lecturas tridimensionales para solucionar el problema de mapeado y localización simultánea. Para finalizar indicar que *Thurn et al.*[24] ha creado mapas tridimensionales de extensas minas.

1.2.1 Algoritmos genéticos de interés histórico

- "Programa de juego adaptativo". Bagley(1967) *et al.*[1]

La primera mención de las palabras "algoritmo genético" y la primera aplicación publicada de un algoritmo genético fueron ambas en la pionera tesis de Bagley(1967). En ese momento había un gran interés en los programas capaces de enfrentarse a juegos y, en ese

ámbito, Bagley diseñó un entorno de pruebas controlables para jugadas basado en el juego del hexapeón.

Bagley construyó algoritmos genéticos para buscar conjuntos de parámetros en funciones de evaluación de juego y compararlas con algoritmos de correlación (procedimientos de aprendizaje modelados a partir de los algoritmos de cambio ponderación de aquella época). Bagley se encontró que los algoritmos de correlación necesitaban una buena coincidencia entre la no linealidad del juego y la no linealidad del algoritmo de correlación.

- “Simulación de la biología celular”. Rosenberg(1967) *et al.*[20]

Trabajando en la misma época que Bagley, Rosenberg (1967) también investigó los algoritmos genéticos en su tesis doctoral. En su estudio, Rosenberg simuló una población de organismos unicelulares simples aunque con bioquímica rigurosa, que incluía una membrana permeable, y estructura genética clásica (un gen, una encima). Independientemente de su énfasis biológico, el trabajo de Rosenberg fue importante para el consecuente desarrollo de algoritmos genéticos en aplicaciones artificiales por su parecido con la optimización.

Otro aspecto interesante de su trabajo es su sistema de cruce adaptativo. En lugar de elegir el lugar de cruce de manera aleatoria uniforme, Rosenberg seleccionó un lugar de cruce determinado por distribuciones de probabilidad.

- “Reconocimiento de patrones”. Cavicchio(1970) *et al.*[3]

En su estudio de 1970, “Búsqueda adaptativa usando evolución simulada”, Cavicchio aplicó los algoritmos genéticos a dos problemas de búsqueda artificial: un problema de selección de subrutinas y un problema de reconocimiento de formas.

En su algoritmo genético, Cavicchio permitió la reproducción y el cruce aproximadamente a como el usado hoy en día. Un mecanismo innovador adoptado en este estudio fue un esquema denominado preselección. En él, una buena generación reemplaza a uno de sus progenitores con la esperanza de mantener la diversidad de la población.

- “Simulación computerizada de una célula viva”. Weinberg(1970) *et al.*[28]

Contemporáneamente con Cavicchio, Weinberg terminaba su tesina. Weinberg sugirió el uso de un algoritmo genético para adaptar los parámetros de un algoritmo genético de menor nivel. Weinberg llamó al algoritmo de mayor nivel programa genético no adaptativo, y al algoritmo de menor nivel programa genético adaptativo (sus parámetros son adaptados). Los parámetros serán pasados al programa genético no adaptativo, que generará y probará poblaciones de proporciones constantes para ser usadas para la subsecuente simulación celular. La tasa de mejora de los datos será pasada también al algoritmo genético de mayor nivel para evaluar la población de los parámetros del algoritmo genético para subsecuentes adaptaciones de alto nivel. Weinberg estaba al tanto de la necesidad de información centralizada en su sistema (1970, p. 101): “Calculamos la utilidad de un programa genético adaptativo indirectamente, usando un juez-deidad, el programa genético no adaptativo.”

1.3 Contenido de la memoria

La memoria está estructurada por capítulos, de manera que, comenzando por el presente capítulo, se introduce al problema de localización global de robots y el método RELF-3D que ha motivado este proyecto, exponiendo las mejoras que se intentará realizar en él mediante programación paralela usando la tecnología CUDA®; en el capítulo 2, *Fundamentos teóricos*, se explicará en profundidad el funcionamiento del método RELF-3D y los fundamentos teóricos con los que se desarrolló su adaptabilidad evolutiva; en el capítulo 3, *Fundamentos de CUDA™*, se relatará cómo la incursión de la arquitectura CUDA® de NVIDIA™ en el mundo de las unidades de proceso gráfico trajo la accesibilidad al desarrollo paralelo al gran público, permitiendo tener la capacidad de un superordenador en cualquier hogar, así como los fundamentos de esta tecnología; en el capítulo 4, *Desarrollo del proyecto,* se entrará en detalle en la metodología seguida para el desarrollo de la función paralela, cuyo objetivo es mejorar las prestaciones de trabajo del método RELF-3D ya mencionado, explicando en distintos puntos la metodología que debe seguirse para el desarrollo en GPUs, los fundamentos algebraicos que serán necesarios, el entorno tridimensional creado para las pruebas posteriores, y el desarrollo del cálculo multihilo; en el capítulo 5, *Resultados experimentales,* apoyaremos el desarrollo relatado en el capítulo anterior con numerosas pruebas, comparativas de tiempo y el estudio del rendimiento mediante la "ley de Amdahl"; en el capítulo 6, *Conclusiones y trabajos futuros,* se analizará si el objetivo del presente proyecto ha sido alcanzado, las ventajas que ello conlleva y se propondrán nuevas alternativas y líneas de desarrollo; por último, en los apéndices A, B y C, se incluirán, respectivamente, el presupuesto para el presente trabajo, la guía de configuración de CUDA® para el usuario, así como el código completo del trabajo desarrollado.

Capítulo 2: Fundamentos teóricos

2.1 Formulación y solución del problema de localización

El problema de la localización puede ser formulado desde un punto de vista Bayesiano, el robot buscará estimar una distribución de sus posiciones *a posteriori* por medio de los datos disponibles. Los datos del sensor pueden ser divididos en dos grupos: $Y_t \equiv \{ z_{0:t} , u_{1:t}\}$, donde "$z_{0:t}$" contiene las medidas del sensor de percepción y "$u_{1:t}$" contiene la información odométrica. La densidad de probabilidad se calculará *a posteriori* de manera recursiva siguiendo dos pasos:

· Regla de medición: La aplicación de la regla de Bayes hasta el último elemento del vector de medición "Y_t" y suponiendo que la observación de "Z_t" es condicionalmente independiente de las mediciones previas dado el estado "X_t", produce:

$$p(x_t \mid Y_t) = \frac{p(z_t \mid x_t)p(x_t \mid Y_{t-1})}{p(z_t \mid Y_{t-1})} \qquad (2.1)$$

· Predicción: El efecto de un intervalo de tiempo en el estado, dadas las observaciones hasta el tiempo *t*, se obtiene observando que:

$$p(x_{t+1} |Y_t) = \int_{R^n} p(x_{t+1} |x_t, u_t)p(x_t|Y_t)d_{x_t} \qquad (2.2)$$

Donde la suposición de que el proceso "X_t" es Markoviano, y entonces "x_{t+1}" es independiente de "Y_t", se ha considerado.

Cada valor de parámetro candidato en $\Re^n$ produce un valor de p(x | y), reflejando la probabilidad posterior de la posición del robot dados los datos hasta el tiempo "*t*". Esta posterioridad necesita ser ponderada de acuerdo a un criterio dado para determinar una $\hat{x}$ estimada del verdadero valor de posición. El algoritmo de localización desarrollado, el cual es probabilístico pero no Bayesiano, se concentra en obtener el mejor "máximo a posteriori" estimador:

$$\hat{x}^{MAP} = \arg\max_x p(x_t|Y_t) \qquad (2.3)$$

Esta aproximación es menos dependiente de supuestos estadísticos, tiene una implementación más simple, es robusta desde un punto de vista estadístico y tiene un menor coste computacional que los métodos Bayesianos.

2.2 Filtro de localización evolutiva

La localización se realiza por medio de un algoritmo de optimización evolutiva de evolución diferencial que, por medio de una función de aptitud, representa el error entre datos reales y estimados de las posibles soluciones. La búsqueda comienza con una población de "N_p" candidatos elegidos al azar, los cuales son introducidos en el módulo de localización y

evolucionan con el tiempo a la mejor solución. Cada candidato es una posible solución al problema de localización global (la posición del robot, con 4D.O.F.). Para cada candidato, su función de aptitud es calculada, comparando datos reales recibidos por medición láser con datos simulados, comienza entonces un bucle basado en selección natural. En él mutarán, se cruzarán, seleccionarán y descartarán los datos, creando con ello una evolución de poblaciones hasta cumplir unas determinadas condiciones de convergencia que terminarán el proceso de localización.

2.2.1 Mutación cruce y selección

La población inicial es perturbada para generar una variación "*v*" acorde a la expresión:

$$\boxed{v = x_a^k + F(x_b^k + x_c^k)} \qquad (2.4)$$

Donde x_a^k, x_b^k, y x_c^k son vectores de parámetros elegidos al azar dentro de la población en la iteración "*k*" (esta "*x*" no es la coordenada cartesiana, si no el elemento de la población definido por 4 coordenadas). "*F*" es un factor real y constante que controla la amplificación de variaciones diferenciales $(x_b^k - x_c^k)$.

Se selecciona un vector de parámetros x_i^k que se perturba con 3 variables aleatorias y el factor cte. *F*, generando el nuevo vector de parámetros "*v*".

Para incrementar la diversidad de una nueva generación, el cruce es introducido. Denotado por $u_i^k = (u_{i,1}^k, u_{i,2}^k, \dots, u_{i,D}^k)^T$, el nuevo vector de parámetros es:

$$u_{i,j}^k = \begin{cases} u_{i,j}^k \; si \; p_{i,j}^k < \delta \\ x_{i,j}^k \; para \; el \; resto \end{cases} \qquad (2.5)$$

Donde $p_{i,j}^k$ es un valor elegido aleatoriamente del intervalo [0, 1] para cada parámetro "*j*" del miembro de población "*i*" en el paso "*k*", y "δ" es la variable de probabilidad y control de cruce.

El nuevo miembro de la población u_i^k es comparado a x_i^k para decidir si el vector u_i^k debe convertirse en miembro de la generación $i + 1$. Si el vector u_i^k produce un mejor valor para el objetivo de la función de aptitud que x_i^k, entonces es reemplazado por u_i^{k+1}; de otra manera, el antiguo valor x_i^k es conservado para una nueva generación.

2.2.2 Mecanismo de thresholding

Su función es rechazar las soluciones que no mejoran la anterior en una magnitud "τ", la cual depende de las variaciones en el ruido de medición. Primero, la diferencia del valor de conveniencia entre el miembro de la población x_i^k y el candidato miembro u_i^k es calculada. Entonces esta diferencia es comparada con un valor de threshold "τ" predefinido en orden de determinar si la mejora mostrada por u_i^k no es causada por el ruido. Si la condición se cumple, el vector objetivo x_i^k es reemplazado por u_i^k en la siguiente generación; de otra manera, x_i^k es

mantenida en la población. Es decir, se puede considerar que la mejora no se debe al ruido y puede ser introducida en la población si la mejora en la aptitud es mayor que la varianza.

2.2.3 Mecanismo de descarte

El método tiene incluido un mecanismo de descarte para evitar la pérdida de velocidad sin perder robustez, especialmente al comienzo de ejecución donde el error tiene más peso. Su cometido es encontrar el peor individuo de una población y sustituirlo por un similar a otro mejor. La manera de conseguirlo es seleccionar un porcentaje de elementos descartables y, para evitar un excesivo parecido en estas sustituciones con el mejor individuo, se selecciona aleatoriamente al que se parecerá entre los mejores. Posteriormente, añadiendo una pequeña variación aleatoria, se adopta como nueva descendencia.

2.2.4 Condiciones de convergencia

La función de aptitud antes de la convergencia vendrá dada por la siguiente expresión:

$$f_0^j\left(x_t^j\right) = \sum_{i=0}^{N_s} \frac{(z_{t,i} - \hat{z}_{t,i}^j)^2}{2\sigma_e^2} = \frac{1}{2}\sum_{i=0}^{N_s} \frac{v_{t,i}^2}{\sigma_e^2} \qquad (2.6)$$

Donde $v_{t,i} = (z_{t,i} - \hat{z}_{t,i}^j)$ representa la discrepancia entre los datos sensoriales observados y las predicciones.

Es interesante que se pudiera encontrar una estimación del valor de la función de coste, ello implica minimizar los efectos que la influencian: el ruido de medición y el error de estimación. Para conseguirlo hay que conseguir una estimación perfecta ya que es inevitable que exista un error asociado al ruido de medición. Sería entonces posible estimar el valor esperado de la función objetivo cuando está cerca del verdadero valor $E(f_0)$. Este sería el valor de la función de aptitud en la posición real.

Si observamos el término $\sum_{i=0}^{N_s} v_{t,i}^2/\sigma_e^2$ de la última expresión, los componentes $v_{t,i}^2/\sigma_e^2$ son variables aleatorias con una distribución normal estándar $-N(0,1)-$, y la suma sigue una distribución de probabilidad Chi-cuadrado con N_s "g.d.l". Esta distribución de probabilidad tiene un promedio de N_s y una varianza $N_s/2$. Por lo tanto, el valor esperado de la función objetivo que estamos tratando de minimizar es

$$E[f_1] = \int_{-\infty}^{+\infty} f_0(v)p(v)dv = N_s/2 \qquad (2.7)$$

Con esto el valor esperado de la función será $N_s/2$ independientemente de los errores de medición o si la posición evaluada era la correcta.

Las condiciones de parada para el algoritmo, aunque no aseguran la convergencia, pueden conducir a buenos resultados en menos tiempo y terminan el proceso de localización:

· Si el número de iteraciones sin cambios en el valor de la función de aptitud de la mejor estimación es mayor que una constante predefinida.

· Si el número de iteraciones sin cambios en el valor de la función de aptitud de la peor estimación es mayor que una constante predefinida.

· Si el número de iteraciones sin cambios en la diferencia entre el valor de la función de aptitud de la mejor estimación y el valor de la función de aptitud de la peor estimación es mayor que una constante predefinida.

2.2.5 Función de aptitud

La función de coste compara los datos del láser obtenidos de la verdadera posición del robot con una simulación estimada de las medidas del láser en ese punto. El algoritmo de localización minimiza el error y evoluciona a la verdadera solución. El método de localización aplica la función de aptitud alineando el escaneo actual (posición estimada) con el escaneo de referencia (posición real). La población evoluciona a la localización correcta porque el error de coincidencia se minimiza.

La elección natural para la función de coste es la suma de la función de errores cuadráticos (L2-norma). Si el vector de observación en el momento "t" es $z_t = (z_{1,t}, \dots, z_{p,t})^T$, las observaciones previstas de acuerdo con la posición estimada del robot son $\hat{z}_t = (\hat{z}_{1,t}, \dots, \hat{z}_{p,t})^T$, y la varianza del error de observación es σ_e^2. Entonces la función de penalización para un único punto x_t^j de la población se puede establecer como

$$L2\left(x_t^j - \hat{x}_t\right) = (z_t - \hat{z}_t)^T (z_t - \hat{z}_t) = \sum_{i=0}^{N_s} \frac{(z_{t,i} - \hat{z}_{t,i}^j)^2}{2\sigma_e^2} \quad (2.8)$$

Factores en la localización global que hacen esta función de aptitud difícil de manejar:

·El rango, la precisión del sensor y el número de sensores limita la posibilidad de distinguir entre diferentes posiciones, lo que lleva a la función de aptitud a un alto número de máximos globales.

·Las similitudes geométricas en el entorno, debidas a la repetición de la distribución espacial, origina la presencia de un alto número de posibles soluciones de la posición del robot.

La función de pérdida, definida de esta manera, nos proporciona un mecanismo de optimización que obtiene una solución con parámetros poco convincentes y, por extensión, un filtro inestable. El filtro evolutivo se puede mover de un mínimo local a otro, originando cambios abruptos en la estimación de posición. Este problema viene del hecho de que no se usa toda la información conocida sobre el sistema en la función. De hecho, sólo se usa la observación modelo para predecir las mediciones del sensor en cada posición, y estas mediciones predichas junto con las medidas actuales son introducidas en la función de pérdida para evaluar la estimación de la posición del robot. La inestabilidad originada por la existencia de múltiples soluciones a la función de pérdida no puede ser solucionada considerando sólo las observaciones del sensor disponibles. Además, el método de mínimos cuadrados no es completamente satisfactorio cuando el modelo de ruido no se conoce exactamente o el modelo está contaminado con otras distribuciones de probabilidad.

El error absoluto (L1-norma) podría ser una medida apropiada en algunas situaciones, ya que presenta un mejor rendimiento con errores grandes, originados en valores atípicos o medidas contaminadas. Si la L1-norma no es derivable, entonces es necesario aplicar métodos de programación lineal para obtener una solución al problema de optimización. Basado en estudios Monte-Carlo, el uso de la L1-norm ha sido recomendado cuando los errores siguen una distribución de Laplace, Cauchy, mezcla de normal y uniforme, o normal contaminada.

Si la función de aptitud seleccionada es la L1-norm, la función para minimizar es dada por la siguiente ecuación:

$$L1\left(x_t^j - \hat{x}_t\right) = |z_t - \hat{z}_t| = \sum_{i=0}^{N_s} \frac{\left|z_{t,i} - \hat{z}_{t,i}^j\right|}{\sigma_e} = \sum_{i=0}^{N_s} \frac{|v_{t,i}|}{\sigma_e} \quad (2.9)$$

Donde $v_{t,i} = (z_{t,i} - \hat{z}_{t,i}^j)$ representa la discrepancia entre los valores observados y los predichos por el sensor de datos.

Capítulo 3: Fundamentos de CUDA™ (Compute Unified Device Architecture)

3.1 Desarrollo histórico de las unidades de proceso

Durante casi 30 años, uno de los métodos para mejorar el rendimiento de los dispositivos comerciales de computación ha sido aumentar la frecuencia a la que el reloj del procesador trabaja. Empezando por los primeros ordenadores personales de principios de los 80, las unidades centrales de proceso, o CPUs, funcionaban con frecuencias de operación en torno a 1MHz. Aproximadamente 30 años después, la mayoría de los procesadores personales tienen velocidades entre 1GHz y 4GHz, aproximadamente 1000 veces más rápido que los del ordenador personal original. Pese a que incrementar la frecuencia del reloj de la CPU no es el único método por el cual se ha mejorado el rendimiento, siempre ha sido una manera segura para aumentarlo. Fuera del mundo comercial, las supercomputadoras han obtenido durante décadas rendimientos excepcionales de las mismas prácticas. Además, junto a las mejoras para un único núcleo de proceso, los fabricantes han conseguido notables rendimientos aumentando el número de procesadores y núcleos presentes.

Sin embargo, las restricciones actuales, así como la proximidad a los límites físicos de tamaño de transistor, están obligando a investigadores y fabricantes a buscar alternativas para obtener mejoras en la velocidad y el rendimiento fuera del tradicional aumento de potencia.

En la búsqueda de potencia de proceso adicional para los ordenadores personales, el desarrollo de los superordenadores lanza una buena cuestión: en lugar de simplemente buscar la mejora de rendimiento de un único núcleo de proceso, ¿por qué no incluir más en los equipos personales? De esta manera, los ordenadores personales podrían continuar su escalada de rendimiento sin necesidad de continuas subidas de velocidad del procesador.

3.2 El auge del cálculo a través de las unidades de proceso gráfico.

3.2.1Breve historia de las GPUs

Es inevitable mencionar la sorprendente revolución ocurrida durante todo el proceso mencionado anteriormente. A finales de los 80 y principios de los 90, el crecimiento de popularidad de los sistemas operativos con interfaz gráfica ayudó a crear un mercado para un nuevo tipo de procesador. Fue en esta época cuando los usuarios comenzaron a adquirir aceleradores de cálculo 2D para las visualizaciones en pantalla.

Aproximadamente al mismo tiempo, en el mundo profesional, la compañía Silicon Graphics popularizó el uso de gráficos tridimensionales en mercados muy diversos. Desde el gubernamental, pasando por aplicaciones de defensa, científicas o de visualización técnica, así como las herramientas para efectos cinemáticos. En 1992 Silicon Graphics publicó su interfaz de programación de hardware al liberar las librerías de OpenGL, pretendiendo así, crear un estándar para el desarrollo de aplicaciones 3D.

Para mediados de los 90, la demanda de aplicaciones que utilizasen gráficos 3D escalaba tan rápidamente, que sentó las bases de dos tipos de desarrollos realmente significativos. Primero, la comercialización de juegos en primera persona, como Doom, Duke Nukem3D y Quake, que disparó el desafío de la creación de entornos visuales tridimensionales cada vez más realistas. Segundo, compañías como NVIDIA®, ATI Technologies o 3dfx Interactive, comenzaron la venta de aceleradores gráficos que fueran lo suficientemente asequibles como para hacerlos atractivos al público general. Estos hechos cimentaron el desarrollo de las tecnologías 3D.

Figura 3.1: Ejemplo de juego con entorno 3D.

Desde el punto de vista de cálculo paralelo, el lanzamiento por parte de NVIDIA® de su serie GeForce 3 en 2001, representa sin duda el avance más importante en la tecnología de las GPU. Fue el primer chip de la industria en implementar el, por aquel entonces, nuevo estándar DirectX 8.0 de Microsoft. Este estándar requería que el hardware compatible contuviera etapas programables para el cálculo de vértices y sombreado de píxeles. Por primera vez, los desarrolladores tenían control preciso sobre los cálculos que serían realizados en las GPUs.

3.2.2Computación GPU temprana

Las GPUs de principios de la década de 2000 fueron diseñadas esencialmente para producir los colores de cada píxel o punto de la pantalla, para ello usaban unidades aritméticas programables conocidas como “pixel shaders”. En general, un pixel shader utiliza la posición (x,y) en la pantalla, así como diversos datos de entrada como colores, texturas y otros atributos, para combinarlos y calcular el color que se mostrará finalmente. Pero dado que los cálculos aritméticos realizados para estos conjuntos de datos son controlados completamente por el programador, los investigadores observaron que de hecho podrían representar cualquier información. Por tanto, si esta información fueran datos numéricos con un significado distinto al código de colores, su programación podría orientarse a obtener cálculos arbitrarios de ellos. Los resultados serían interpretados por la GPU como el color que finalmente tendría un píxel, pero realmente sería el cálculo final de lo que el programador hubiera enviado a la GPU. Durante las investigaciones se consiguió traer de vuelta los resultados sin que la GPU realizase modificaciones por pantalla, esencialmente, se engañó a la

GPU para realizar tareas ajenas al renderizado programándola de manera que parecieran cálculos gráficos.

Los resultados obtenidos con este truco fueron muy prometedores, y auguraban un gran futuro al cálculo aritmético a través de la GPU. Sin embargo, era un modelo demasiado restrictivo para el gran público: los recursos eran muy limitados, ya que los datos sólo podían ser transmitidos por dispositivos dedicados a colores y texturas; existían limitaciones al cuándo y cómo el usuario podía escribir en la memoria, por lo que los algoritmos que necesitaban esta capacidad no podían ser ejecutados en la GPU; cuando el programa realizaba cálculos incorrectos, bloqueaba el equipo, o terminaba su ejecución con error, no existía una manera de realizar un seguimiento de los errores del código ejecutado en la GPU. Por último, por si todo esto fuera poco, era necesario el aprendizaje de OpenGL o DirectX para realizar la programación, ya que eran el único medio disponible. Todo esto suponía afrontar demasiadas restricciones respecto a la programación tradicional como para tener una amplia aceptación.

3.3 Alternativa al modelo secuencial: CUDA™

Es importante recalcar que las GPUs son diseñadas como motores de cálculo numérico, y no realizan eficientemente algunas tareas para las cuales han sido diseñadas las CPUs; por tanto, es de esperar que las aplicaciones deberían usar tanto la CPU como la GPU, ejecutando las partes secuenciales en la CPU y las partes de intensivo cálculo numérico en la GPU. Esta es la razón por la que el modelo de programación CUDA™ fue introducido por NVIDIA®, ya que está específicamente diseñado para soportar la ejecución conjunta CPU-GPU en las aplicaciones.

Debemos mencionar que el rendimiento no es el único factor decisivo en todos estos acontecimientos; el coste del desarrollo de software debe estar justificado con un gran número de consumidores, por lo que se deben crear aplicaciones capaces de ejecutarse en procesadores presentes en un número suficientemente grande de equipos. Este era tradicionalmente uno de los problemas de los sistemas de cálculo paralelo: su presencia era ínfima en comparación con los microprocesadores de propósito general.

Un último factor de consideración para la selección de procesador para la ejecución de aplicaciones de cálculo numérico es el cumplimiento del estándar de coma flotante del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE). Este estándar hace posible obtener resultados predecibles entre procesadores de diferentes vendedores.

No sería hasta cinco años después de la comercialización de las GeForce 3, en noviembre de 2006, cuando todo esto cambió; se reveló la primera GPU con DirectX 10 de la industria: la GeForce 8800 GTX (o G80 por simplicidad). Esta GPU fue también la primera que se fabricó con la nueva arquitectura CUDA™ de NVIDIA® y el estándar del IEEE. Esta arquitectura incluyó numerosos componentes completamente nuevos diseñados estrictamente para cálculos en la GPU, cuyo objetivo era aliviar las numerosas limitaciones existentes previas que impedían orientar los procesadores gráficos hacia el cálculo generalizado. Con ella llegaron de manera masiva las GPUs multi núcleo al mercado de los PC, habiéndose vendido cientos de millones de GPUs hasta la fecha, lo que ha hecho factible el desarrollo del cálculo paralelo en el mercado de manera masificada.

Desde 2007 todo comenzó a cambiar. En el G80, y en los sucesivos chips para cálculo paralelo, los programas CUDA™ utilizaron una nueva interfaz de propósito general que satisfacía las peticiones de los programas paralelizados por esta nueva arquitectura. Por otra

parte, también el software se rehízo, de manera que los programadores pudieran utilizar herramientas más familiares de programación en entornos de C/C++, evitando necesitar las librerías de programación gráfica para aplicaciones de cálculo.

3.4 Arquitectura de una GPU moderna

¿Qué hace más aptas a CPUs y GPUs para cada trabajo? La respuesta radica en la filosofía de diseño entre ambos tipos de procesadores. Esta diferencia de diseño está ilustrada en la figura 3.2.

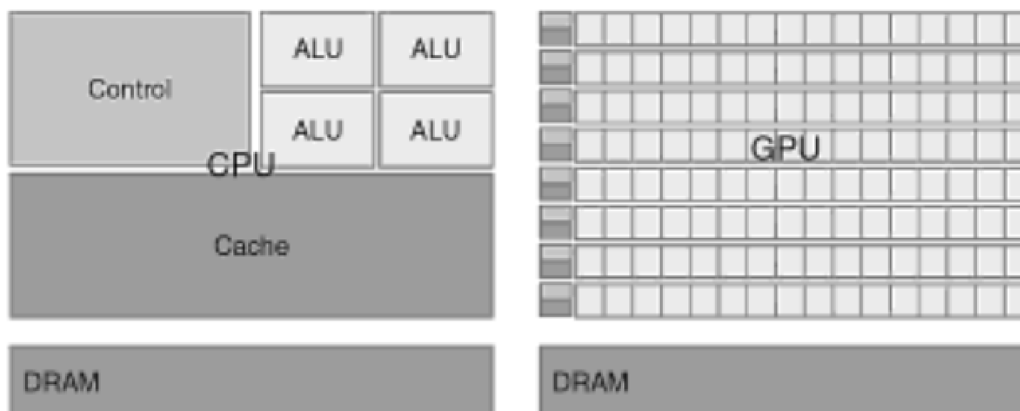

Figura 3.2: Diferencias de construcción.

El diseño de una CPU está optimizado para el rendimiento del código secuencial, haciendo uso de una sofisticada lógica de control para permitir a las instrucciones de un único hilo de ejecución trabajar en paralelo o incluso fuera de la secuencia mientras mantienen la apariencia de una ejecución secuencial, llegando incluso, desde 2009, a fabricarse CPUs con cuatro núcleos especialmente enfocados en aumentar este rendimiento secuencial. Además se introduce una memoria caché de gran tamaño para reducir las latencias de acceso para instrucciones y datos en aplicaciones de gran complejidad. Por otra parte, el ancho de banda de las CPUs actuales para la memoria DRAM, está entorno a los 50GB/s, siendo difícil su crecimiento, ya que los requerimientos de regulación de acceso a esta memoria para sistemas operativos, software y dispositivos de entrada y salida, tienen que satisfacer unas características heredadas históricamente harto complicadas de modificar.

La filosofía de diseño de las GPUs está condicionada por la vertiginosa industria de los videojuegos, la cual ejerce una enorme presión económica por medio de la búsqueda de la habilidad de desempeñar una ingente cantidad de cálculos de coma flotante por fotograma de vídeo en los juegos más avanzados. Esta demanda motiva a los fabricantes de GPUs a buscar maneras de maximizar el presupuesto para tamaño y potencia de sus chips dedicados a esta tarea. La solución más adoptada actualmente es optimizar la ejecución a través de una enorme cantidad de hilos. El hardware se aprovecha de la ejecución de esta gran cantidad de hilos buscando tareas que realizar mientras algunos de ellos esperan para acceder a la memoria de mayor latencia, minimizando así la lógica de control requerida para la ejecución de cada hilo. Las pequeñas memorias caché presentes ayudan a controlar los requerimientos de ancho de banda de las aplicaciones, de manera que si múltiples hilos necesitan acceder a los mismos datos de la memoria, no es necesario que todos accedan a la memoria DRAM. Como resultado, un mayor área del chip es dedicado a los cálculos de coma flotante. Sin embargo, el ancho de banda de acceso a esta memoria DRAM es también mayor que en los chips de las CPUs, siendo

capaces de manejar datos hasta una velocidad de 85 gigabytes por segundo dentro y fuera de ella en el caso del chip G80 original, llegando hasta los 672 GB/s de la actual NVIDIA® Titan Z basada en la arquitectura Kepler.

3.4.1 Arquitectura NVIDIA® Kepler

El proyecto que nos atañe ha sido realizado usando la GPU GeForce GTX 670, basada en la arquitectura Kepler-GK104 de NVIDIA®, la más novedosa hasta la fecha, y que ha sido desarrollada a partir de las anteriores arquitecturas Tesla y Fermi. Es por esta razón que se desarrollará en este apartado tanto su construcción como funcionamiento a partir del primer producto introducido de la gama: la GeForce GTX 680.

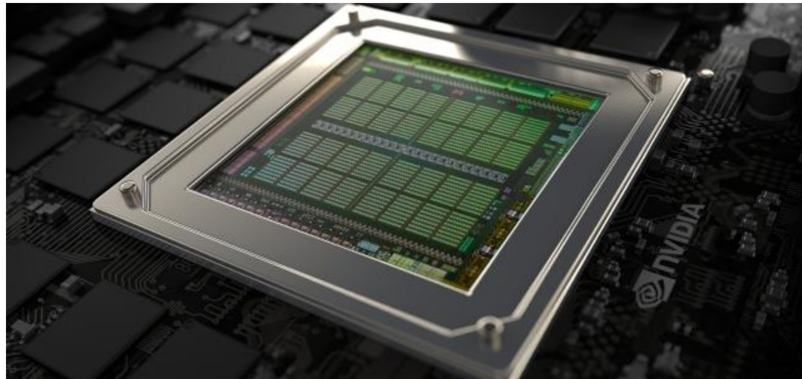

Figura 3.3: Chip de NVIDIA®.

La introducción de esta arquitectura implica que los desarrolladores de videojuegos podrán incorporar mayores niveles de complejidad geométrica, simulaciones físicas, procesado estereoscópico 3D… Dadas las características de CUDA™, es evidente que este aumento de capacidad de cálculo también tiene repercusión en los recursos disponibles para la programación de propósito general.

Para el desarrollo de esta arquitectura, observable en la figura 3.4, los ingenieros de NVIDIA® han buscado mantener el liderazgo de la compañía en términos de rendimiento gráfico añadiendo a ello un mayor rendimiento por vatio de potencia. Para ello se han rediseñado los anteriores multiprocesadores de transmisión (SMs), dando lugar a los nuevos “SMX”, reduciendo el consumo de potencia de la GPU a través de diversos cambios.

La configuración de fabricación incluye cuatro “Clúster de Procesado Gráfico” (GPCs); ocho de los mencionados SMX, que trabajan en la misma frecuencia que el procesador gráfico en lugar del doble como ocurría con los anteriores; sin embargo con 1536 núcleos CUDA™ en la Kepler-GK104, la GTX 680 proporciona el doble de rendimiento por vatio de la anterior Fermi-GF110 (esta arquitectura incluía 512 núcleos CUDA™); y finalmente cuatro controladores de memoria.

De nuevo remitiendo a la figura 3.4, puede observarse que cada GPC tiene un motor de rasterización dedicado, cuya función es, básicamente, tomar una imagen definida como un vector y transformarla en píxeles o puntos. Adjunto a cada controlador de memoria, está una caché L2 de 128KB y ocho unidades ROP (cada una de estas unidades ROP procesa una única

muestra de color), dando lugar a un total de 512KB de caché L2 y 32 ROPs. Finalmente, el PE-2.0 (Polymorph Engine), maneja la búsqueda de vértices, teselación (o mallado de polígonos para una mejor representación 3D), transformaciones vectoriales, configuración de atributos y salida de transmisión.

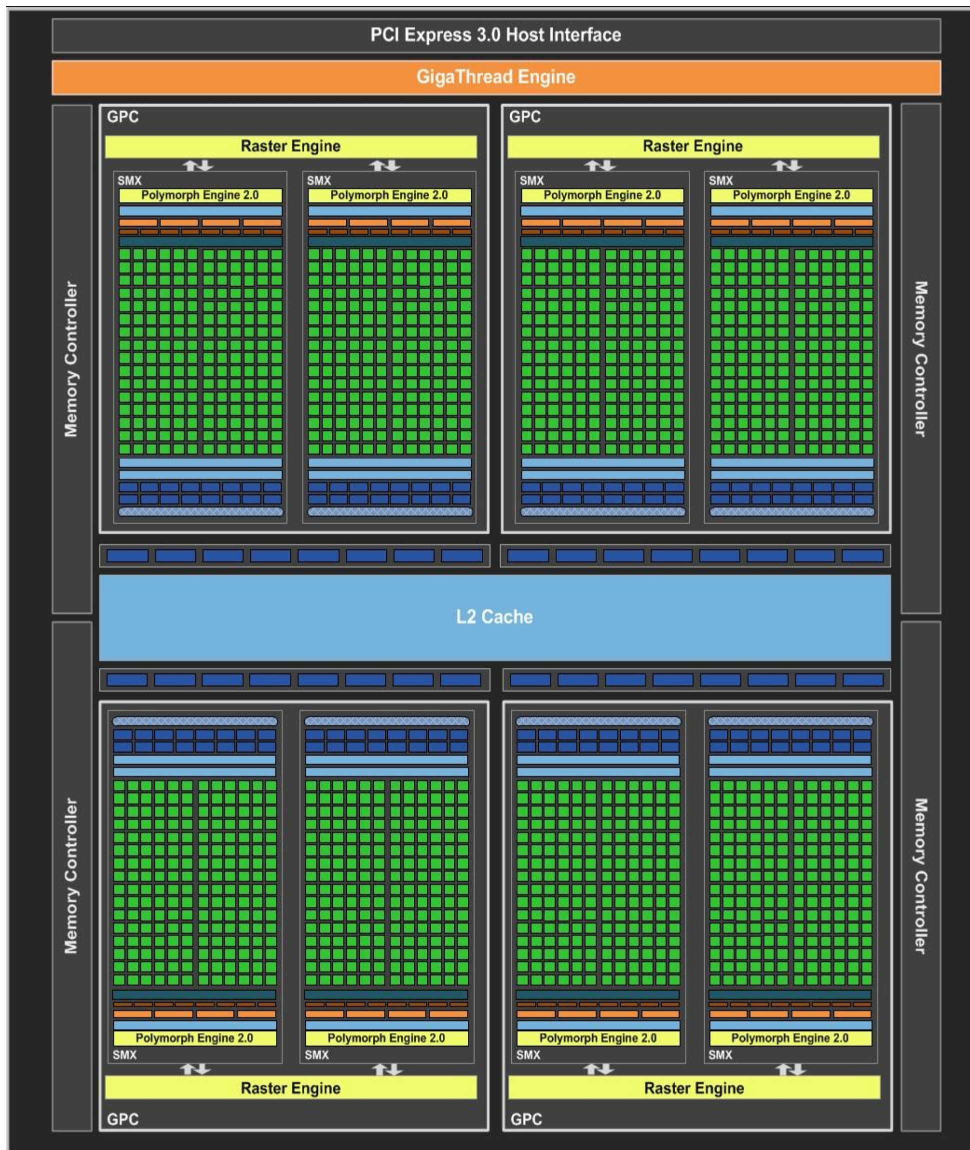

Figura 3.4: Arquitectura CUDA.

3.5 Relación velocidad y cálculo paralelo

Como ya se ha mencionado anteriormente, la principal motivación para la programación paralela es incrementar la velocidad de las aplicaciones de cara a las futuras generaciones de hardware. Es lógico preguntarse por qué las aplicaciones seguirán necesitando estos incrementos de velocidad, ya que muchas de las actuales parece que ya se ejecutan lo suficientemente rápido; pero cuando una aplicación es adecuada para la ejecución paralela, una buena implementación para la GPU puede llegar a conseguir acelerarla hasta 100 veces respecto a la original. Si la aplicación incluye paralelismo entre los datos que maneja, es

fácil acelerarla 10 veces con tan sólo unas pocas horas de trabajo. Puede observarse en la figura 3.5 la diferencia de operaciones de coma flotante (FLOPS)

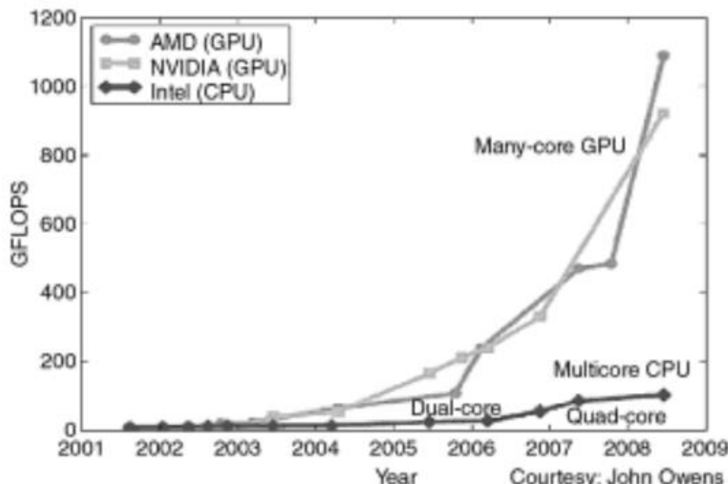

Figura 3.5: Comparativa de operaciones de coma flotante.

3.5.1 Aplicaciones

3.5.1.1 Imágenes médicas

La empresa "TechniScan Medical" ha desarrollado un sistema para la creación de imágenes tridimensionales por medio de ultrasonidos, de manera que se pueda usar como técnica menos invasiva para las pruebas de cáncer de mama, basadas éstas en mamografías por rayos X, y en muchos casos, biopsias adicionales de las zonas de riesgo. Sin embargo, este método no se comenzó a poner en práctica hasta 2010, dado que requería unos tiempos de computación prohibitivos y un equipo muy caro. No ha sido hasta ese año cuando su sistema "TechniScan Svara" se pudo poner en funcionamiento, el cual realiza una exploración por ultrasonidos durante 15 minutos para posteriormente, usando dos procesadores NVIDIA® Tesla C1060 basados en la tecnología CUDA™, procesar 35Gb de datos en aproximadamente 20 minutos para crear una imagen tridimensional del pecho.

3.5.1.2 Dinámica de fluidos

El enrevesado movimiento de aire y fluidos dentro de rotores implica una compleja formulación para su modelado y el posterior diseño de susodichos rotores. Sólo los superordenadores al alcance de unas pocas compañías podían realizar las simulaciones necesarias para estos cálculos. Esto ha cambiado desde que los investigadores de la universidad de Cambridge, tras años de estudio de la arquitectura CUDA™, han desarrollado métodos de cálculo y simulación de fluidos usando pequeños clúster de GPUs. En poco tiempo han superado la capacidad de los superordenadores tradicionales para este tipo de cálculos y simulaciones y, además, han demostrado que dadas las características del mercado de procesadores gráficos, la relación de coste de cómputo es infinitamente menor.

3.5.1.3 Ciencia medioambiental

Los detergentes y productos de limpieza son algo cotidiano en el mundo actual. Su desarrollo se basa en el uso de emulsionantes que determinan la capacidad de limpieza y textura de ellos. Por desgracia también se les suele identificar como el componente más devastador en términos medioambientales, lo que implica que necesitan exhaustivas pruebas en laboratorio que acarrean numerosas combinaciones de materiales e impurezas, lo que, evidentemente, es un proceso largo y costoso. La universidad de Temple, de manera conjunta a la empresa dedicada a este sector, Procter&Gamble, trabajan en simulaciones de suciedad, agua y materiales por medio de ordenadores, que sirven no sólo para acelerar los cálculos de laboratorio, si no que amplían la cantidad de variantes que se pueden analizar. Para sus investigaciones utilizaron un método acelerado por cálculo en la GPU por medio de dos procesadores gráficos Tesla de NVIDIA®, con ello han conseguido rendimientos equivalentes a la CPU de 128 núcleos de Cray XT3 y a los de el superordenador BlueGene/L de 1024 CPUs de IBM. En investigaciones posteriores incrementaron el número de GPUs Tesla obteniendo un rendimiento de cálculo para las simulaciones 16 veces superior a las plataformas citadas.

Todo lo tratado en este capítulo en conjunto proporciona una idea de la capacidad y el horizonte de posibilidades que abrió desde 2007 CUDA™.

Capítulo 4: Algoritmo RELF-3D acelerado

En este capítulo se explicarán los conceptos básicos, así como la metodología seguida para el desarrollo del proyecto. Se analizarán posibles alternativas y los obstáculos encontrados.

4.1 Conceptos básicos de programación CUDA™.

Para comenzar la explicación del desarrollo del proyecto es necesario aclarar ciertos puntos que permitirán comenzar a visualizar la manera de proceder en las aplicaciones paralelizadas:

- En el capítulo anterior se trató la evolución de las GPU y su arquitectura, quedando claro que es un dispositivo añadido dentro del ordenador distinto de la CPU. Las actuales GPGPUs (General Purpose GPU), en la gran mayoría de casos, están conectadas por una interfaz de alta velocidad conocida como PCIe (Peripheral Component Interconnect express), a través de la cual se realizan tanto las transferencias de datos como de comandos. Dependiendo de las características del equipo y la placa base pueden llegar a añadirse hasta cuatro GPGPUs, siendo cada una de ellas dispositivos que trabajan de manera asíncrona con el procesador. Esto significa que podrían estar trabajando tanto el procesador y hasta cuatro unidades gráficas en conjunto, realizando cualquier tipo de cálculos. La memoria que utilizan está completamente separada para dispositivos y procesadores, teniendo un ancho de banda muy distinto en cada caso como se ha explicado en el capítulo anterior. Sin embargo, es posible realizar transferencias de datos entre ellas. Para ello se utiliza el comando "***cudaMemcpy()***", cuyo uso es similar al "***memcpy***" disponible en C.

- Los programas realizados con CUDA™ utilizan "***kernels***". Se trata de subrutinas que se ejecutan en el dispositivo CUDA™ y que se llaman desde el código que se está ejecutando en el host. Debemos resaltar que no se trata de funciones normales, ya que no pueden devolver un valor, si no que sirven para gestionar el hardware. Un kernel se define con la declaración "***__global__***" que le indica al compilador que se trata de un kernel, y que puede ser llamado por el procesador host mediante el comando "***kernel<<<bloques,hilos>>>***". Las llamadas a los kernels son *asíncronas*, lo que significa que el host realiza la cola de ejecuciones del programa y sigue trabajando mientras la GPU trabaja. Es por esta razón que el kernel no puede devolver un valor como una función. Existen maneras de hacer que el host espere al final de la ejecución en la GPU, mediante "***cudaThreadSynchronize()***" o por la transferencia de datos entre memorias usando el ya citado "***cudaMemcpy()***", ya que esta última ejecuta internamente también la anterior sin necesidad de escribir ambas.

- La unidad básica de trabajo en una GPU es el hilo. Es importante recalcar que desde el punto de vista del software, cada hilo está separado del resto y actúa como si tuviera su propio procesador, con registros e identidad separada, aunque realmente se ejecuta en un entorno de memoria compartida. Las características del hardware son

las que limitan el número de hilos que se pueden ejecutar de manera concurrente. Los hilos se ejecutan en agrupaciones dentro de bloques y, a su vez, los bloques se agrupan formando un "***grid***".

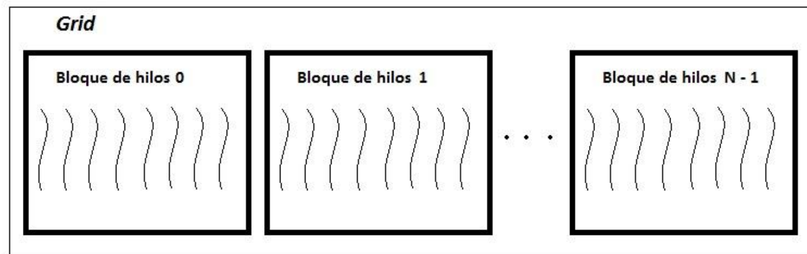

Figura 4.1: Representación Grid.

- La mayor región de memoria compartida en la GPU se llama "***memoria global***". Se mide en gigabytes de RAM y es donde residen la mayoría de datos que manejan las aplicaciones. El acceso a ella es rápido (160-200GB/s) y da una gran capacidad de transferencia y rendimiento, sin embargo, está muy lejos de los teraflops de datos que llega a manejar una GPU. Debido a este cuello de botella, es muy importante reusar los datos dentro de la GPU para conseguir el máximo rendimiento.

4.1.1 Las 3 reglas de la programación de GPGPUs.

Regla 1: Llevar los datos a la GPU y mantenerlos en ella. Si el acceso a la memoria dentro de la GPU ya supone un punto de conflicto por la ralentización que supone la transferencia de datos a más de 160GB/s, el problema se incrementa más de 20 veces si la transferencia de datos se realiza con el host de por medio, ya que el bus PCIe direccional proporciona tan sólo 8GB/s en su versión 2.0 y 16GB/s en su versión 3.0.

Regla 2: Proporcionar a la GPGPU suficiente trabajo para hacer. Cada kernel debería realizar alrededor de 1 millón de operaciones de coma flotante para evitar el desperdicio de ciclos de cálculo, ya que supone mayor retraso el comienzo de cada kernel que el número de operaciones.

Regla 3: Reutilizar datos dentro de la GPGPU. Para evitar las limitaciones de ancho de banda de la memoria se deben explotar al máximo los recursos disponibles internamente en la GPU. Esta norma es, evidentemente, aplicable también para los procesadores convencionales.

Estas tres reglas orientan la manera de enfocar el diseño de un programa y deben cumplirse en la medida de lo posible para maximizar el rendimiento. En el presente proyecto se ha conseguido mantener los datos y reutilizarlos de manera eficiente como indican las reglas 1 y 3. La regla 3 se ha cumplido en la medida de lo posible, ya que la cantidad de cálculos que se realizarán es menor que la capacidad de la GPGPU utilizada.

4.1.2Conceptos algebraicos

Por las características de la reserva de memoria, tanto para el código a ejecutar en el host como en la GPU, es necesario aclarar su funcionamiento, ya que es crucial en el desarrollo del proyecto. En términos de gestión de memoria, es común en los lenguajes de programación la reserva de variables por medio de declaraciones que definen su tipo, ya sea entero, carácter,

etc. El lenguaje C añade otra manera de declarar las variables, y es mediante el uso de punteros. Los punteros permiten hacer reservas dinámicas de memoria, declarando el tipo de variable y, en caso de ser una matriz, las dimensiones. La parte compleja radica en que las posiciones de memoria trabajan como listas, es decir, convierten los datos que va a guardar cada posición de la memoria reservada en una sucesión. Esta conversión de coordenadas de datos a una sucesión, especialmente crucial cuando se trabaja con matrices, es en la que se basan tanto las listas de memoria como el paralelismo entre hilos que utiliza la programación con CUDA™.

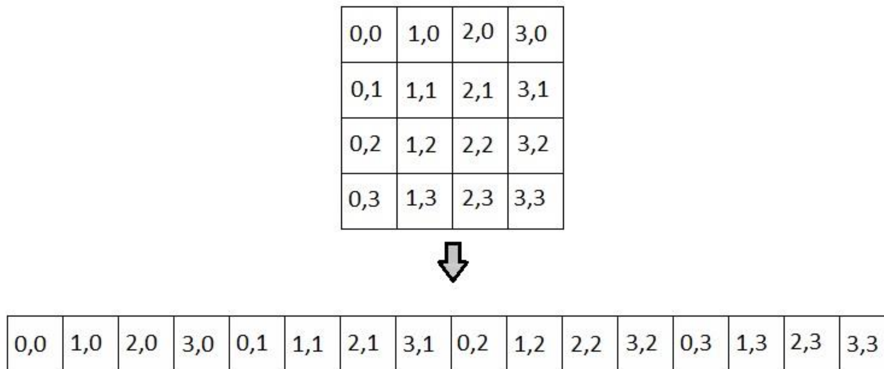

Figura 4.2: Conversión matriz bidimensional en direccionamiento lineal.

Para ilustrar el concepto, tanto de conversión a lista como de paralelismo entre datos, utilizaremos una multiplicación entre matrices. Supongamos que tenemos dos matrices $M[i][j]$ y $N[j][i]$. Queremos obtener con ellas otra matriz $P[i][i]$ cuyos elementos son el resultado de la multiplicación entre las matrices M y N. Siguiendo el álgebra de matrices es fácil saber que cada elemento de la matriz P se obtiene multiplicando los elementos de la fila correspondiente de M por los elementos de la columna de N y realizando su posterior suma.

Estas matrices, dadas las características del lenguaje C, independientemente de si se tratan de variables declaradas normalmente o declaradas como punteros, en memoria se guardan de manera secuencial. La diferencia radica en que las variables declaradas como punteros es el usuario el que debe gestionarlas completamente y, en el otro caso, el compilador se ocupa de su gestión en la memoria.

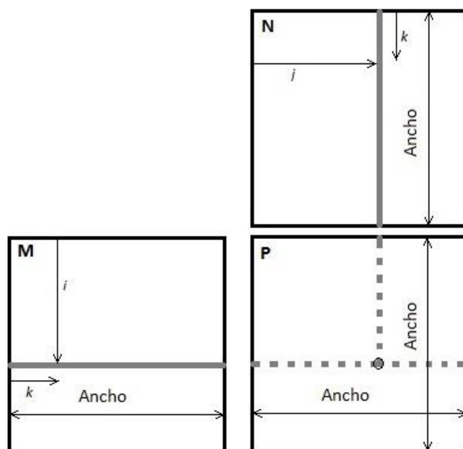

Figura 4.3: Producto matricial.

El resultado de cada elemento de P, se rige por la ecuación:

$P[i * ancho + j] = \sum_{k=0}^{ancho}(M[i * ancho + k] + N[k * ancho + j])$ (4.1)

Este cálculo puede extenderse a matrices N dimensionales, ya que para ello basta con ampliar la fórmula añadiendo las dimensiones adicionales:

Posición de la coordenada X + (Número de coordenadas X · Posición de la coordenada Y) + (Número de coordenadas X · Número de coordenadas Y · Posición de la coordenada Z) + …

Es muy importante dejar claro estos conceptos, ya que se usan a lo largo del proyecto para indexar matrices unidimensionales que representan matrices N dimensionales, y además, es la manera en que son indexados los hilos paralelos que se ejecutan en la GPU mediante CUDA™.

4.1.3 Configuración de la GPU

4.1.3.1 Creación del kernel

En CUDA™, una función kernel especifica el código que será ejecutado por los hilos durante la fase paralela. La notación específica de CUDA™ para declarar una función paralela, que se ejecutará en la GPU, como accesible para las funciones dentro del código del host será usando la palabra reservada por las librerías de CUDA™ "**__global__**", que deberá situarse precediendo la declaración de la función kernel creada. Existen otras dos palabras reservadas en CUDA™ que pueden preceder a una función. La palabra reservada "**__device__**" indica que se trata de una función que se ejecutará en la GPU y que sólo puede ser llamada desde otra función que está siendo ejecutada en la GPU. La palabra reservada "**__host__**" indica que se trata de una función que se ejecutará en el host y que sólo puede ser llamada por el host. Una función declarada de esta última manera se trata de una función como se conoce tradicionalmente en C. Todas las funciones creadas en el código del host usando las librerías de CUDA™ son consideradas así por defecto si no llevan las otras palabras reservadas. Adicionalmente en la declaración de la función se añadirán los parámetros de la misma manera que en otras declaraciones de funciones en lenguaje C.

Cuando se ejecuta la función kernel, todos sus hilos ejecutarán el mismo código definido en la función, por lo que es necesario tener un mecanismo para distinguirlos, escribiendo así una única función genérica que cada hilo ejecutará de manera diferente. Para ello, existe la palabra reservada **"threadIdx"**. Los hilos pueden ser programados para tener una configuración tridimensional dentro del bloque que los contiene y la manera de identificar los N hilos será usando esta palabra clave con sus coordenadas: "**threadIdx.x**", "**threadIdx.y**", "**threadIdx.z**"; siendo la primera coordenada el valor 0 y la última N-1. Usando estos identificadores podemos crear distinciones dentro de un mismo código. El máximo actual de hilos que puede crearse en un bloque es de 1024; esta limitación es intencionada dadas las características del hardware actual, siendo posible que en el futuro sea aumentada, pero actualmente la combinación de "$X * Y * Z$" no puede rebasarla. Es importante recalcar que esta combinación tridimensional está orientada a facilitar las necesidades del programador, sin embargo, los hilos pese a ser indexados con tres coordenadas, computacionalmente son

organizados como una lista o matriz unidimensional, por lo que para ser usados de nuevo referimos al ejemplo de la figura 4.2 y la ecuación 4.1.

Los hilos se engloban dentro de bloques. De manera similar, existe una palabra reservada para manejar los bloques. En este caso es “**blockIdx**”. Los bloques también pueden ser declarados de manera tridimensional en el grid que los engloba y, también de manera similar a la definición de hilos, la manera de identificar sus coordenadas es “**blockIdx.x”** “**blockIdx.y**” “**blockIdx.z**”, siendo también las coordenadas de 0 hasta N-1 bloques declarados. De nuevo, los bloques son organizados también de manera unidimensional, siendo aplicables otra vez el ejemplo de la figura 4.2 y la ecuación 4.1.

Por último queda por explicar el grid, que es la parte que engloba los bloques que contienen los hilos. Cada vez que un kernel es lanzado, se ejecuta un grid de hilos paralelos organizados de la manera descrita, llegando a manejarse millones de hilos en la GPU. Todos los bloques dentro del grid deben estar organizados de la misma manera. La manera de declarar estas dimensiones será usando el comando “**dim3”** con el que se especificaran las tres dimensiones del grid y/o bloques. Como ya se ha descrito anteriormente, la llamada al kernel dentro del código del host se realiza con el comando mi_kernel<<<bloques,hilos>>>(param1, param2,…), los parámetros adicionales son las variables que se copiaran entre la memoria del host y de la GPU y para los que previamente habremos reservado memoria.

Las necesidades de programación, en cambio, pueden requerir una configuración distinta. Un ejemplo de configuración de las dimensiones del grid y de sus bloques sería:

```
dim3 dimBlock(32,16,2)

dim3 dimGrid(5000,3000,200)

mi_kernel<<< dimGrid , dimBlock >>>(param1, param2, …)
```

Dentro de los paréntesis angulados definiremos la cantidad de bloques e hilos que contendrán estos. No es necesario usar la definición mediante el comando “**dim3**”, ya que se puede indicar una cantidad para ambos directamente en los paréntesis angulados al efectuar la llamada, y se ejecutará de manera interna por el compilador, definiendo bloques e hilos de manera unidimensional. De la manera mostrada en el ejemplo, se ejecutaría una llamada a un kernel llamado “**mi_kernel**” que crea un grid tridimensional de $5000 * 3000 * 200$ bloques, cada uno de ellos con $32 * 16 * 2$ hilos.

Los parámetros que se pasarán al hacer la llamada de la función kernel serán las variables que contendrán los datos que usarán los hilos paralelos. Estas variables y su contenido deben tener un espacio reservado en la memoria de la GPU. El manejo de estos datos y la memoria se especifica más adelante.

En el caso del grid también existen limitaciones por razones de hardware para la cantidad de bloques que se pueden crear en su interior. En este caso, para la GPU usada (670 gtx), las dimensiones máximas que puede manejar son $2147483647 * 65535 * 65535$. El total de hilos paralelos que podría manejar esta GPU sería más de $9{,}44 * 10^{21}$.

Es de suponer la duda que esta cantidad de hilos plantea. La posibilidad de tener hilos que puedan terminar unos antes que otros, con los posibles errores o ralentizaciones que esto puede conllevar. La arquitectura de CUDA™ plantea una solución para esto por medio de envoltorios que crean agrupaciones de 32 hilos paralelos cada uno. Durante la ejecución se controlará que estos envoltorios lleven un funcionamiento acorde y que todos ellos realicen las operaciones a la vez, haciendo esperar a los que pudieran terminar antes. Por tanto es interesante declarar dimensiones de hilos y bloques con esto en mente. Adicionalmente, existe un comando para sincronizar todos los hilos en ejecución dentro de un bloque y hacer esperar a los que han terminado antes, de esta manera todos terminan antes de pasar a la siguiente fase. Con esto se consigue que el código se ejecute de manera controlada por el programador. El comando capaz de realizar esto es "**__syncthreads()**". Este comando debe usarse con cuidado, ya que todos los hilos del bloque deben ejecutarlo de la misma manera. Si se usase la construcción "**if-then-else**" usando el comando "**__syncthreads()**" tanto dentro de la parte "**then**" como de la parte "**else**", si distintos hilos entra en distintas partes, cada hilo tendría una barrera de sincronización distinta y se esperarían mutuamente de manera indefinida y sin condición de salida.

4.1.3.2 Gestión de memoria

La función "**cudaMalloc()**" puede ser llamada desde el código que ejecuta el host para asignar una parte de la memoria global de la GPU al elemento que se quiere transmitir para ser utilizado. Este comando presenta una evidente similitud con el comando "**malloc()**" del lenguaje C. Este parentesco es intencionado, ya que CUDA™ es una extensión de las librerías del lenguaje C para mantener una interfaz de programación lo más similar posible y, así, minimizar el tiempo de aprendizaje de nuevos comandos por parte de los programadores. El formato, sin embargo, difiere del original de C ya que la función **malloc()** de C devuelve un puntero al objeto asignado; la función **cudaMalloc()** requiere como primer parámetro la dirección de la variable puntero reservada, transformándola mediante conversión de tipo en un puntero "**void**", convirtiéndose así en el puntero que apunta a la región de la memoria global de la GPU asignada para copiar la variable que deseemos tener allí; el otro parámetro requerido por la función, es un valor entero que indique el tamaño de memoria que se debe reservar.

Un ejemplo de uso de "**cudaMalloc()**" sería:

```
float *M;

int size = X*Y*sizeof(float);

cudaMalloc((void**)&M, size);

cudaMemcpy
```

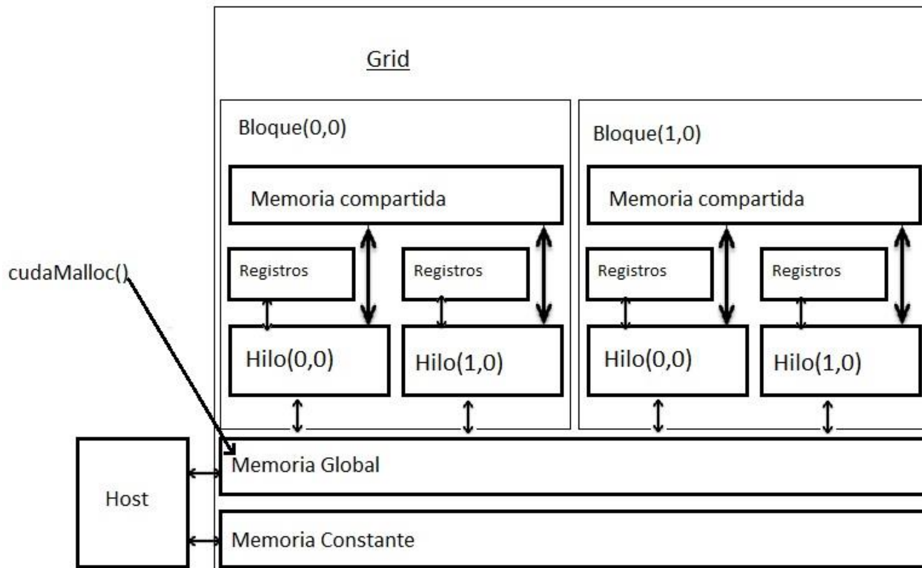

Figura 4.4: Asignación de memoria.

Una vez que el programa ha reservado la memoria global necesaria dentro de la GPU para los elementos que vamos a usar, puede hacerse la transferencia de datos desde el host a la GPU. De nuevo esto se hace con una de las funciones disponibles dentro de la API de CUDA™, en este caso "**cudaMemcpy()**", que se ocupa de la transferencia de datos entre memorias. En este caso el comando "**cudaMemcpy()**" requiere cuatro parámetros: El primer parámetro es un puntero a la localización de destino para la operación de copia; el segundo parámetro es el elemento que será copiado; el tercer parámetro especifica el número de bytes que serán copiados; el cuarto parámetro será los tipos de memoria involucradas en el proceso: entre memorias del host, desde el host a la GPU, desde la GPU hacia la memoria del host o entre memorias de la GPU. Es importante resaltar que esta copia de memoria sólo puede realizarse de la manera definida. El comando "**cudaMemcpy()**" no puede usarse para realizar copias entre distintas GPUs si el sistema es multi-GPU. En este caso las copias de memoria deben pasar por la memoria del host como si de un puente se tratase. Por último, la transferencia se realizará de manera asíncrona.

Un ejemplo de uso sería:

```
float M, N, P

int size = D*D*sizeof(float)

float* Md, Nd, Pd
```

//Transferencia de M y N a la memoria de la GPU

```
cudaMalloc((void**) &Md, size)

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice)

cudaMalloc((void**) &Nd, size)

cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice)
```

//Reservar P en la GPU

```
cudaMalloc((void**) &Pd, size)
```

→Aquí se realizaría la llamada al kernel, que a su vez realizaría las pertinentes operaciones

//Transferencia de P desde la GPU al host

```
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost)
```

4.1.3.3 Gestión de errores

Cualquier programa debe ser capaz de gestionar errores, ya que pueden surgir numerosas situaciones inesperadas fuera de la compilación. Por ello, y dadas las características de ejecución con CUDA™ ya mencionadas, es crucial una buena capacidad de gestión de excepciones. En el presente proyecto se han dado situaciones durante el desarrollo de su programación en las que, sin ellos, encontrar el fallo habría sido muy difícil, ya que el carácter asíncrono de la ejecución del programa permite que un error en la parte ejecutada en la GPU no influya en la ejecución del resto del código del host. Será evidente si se produjera dado que los resultados obtenidos o el comportamiento del programa serán extraños, lo que no será evidente es el punto en el que ocurre el fallo, ya sea por defecto de programación o por un error inesperado durante la ejecución, cosa que puede suceder en cualquier programa y lenguaje aunque todo el código escrito sea correcto, pero un fallo del propio equipo durante la ejecución interrumpa el programa. Para ello se incluyen en puntos críticos esta gestión de errores y así poder manejarlos, investigarlos y, en la medida de lo posible, prevenirlos o corregirlos.

La función "**cudaGetLastError()**" devuelve el último error que se ha producido durante cualquiera de las llamadas en tiempo de ejecución para las funciones específicas de CUDA™ y la reinicia como "**cudaSuccess**". Esto quiere decir que la función devuelve por defecto "**cudaSuccess**", interpretando que la ejecución ha sido correcta y, si se produce un error, no lo hará, teniendo que usar otra función para interpretar el código de error que devolverá.

Cuando se produce este error, una manera de investigarlo es usar la función "**cudaGetErrorString()**", con ella transformaremos el código del error en una información que el usuario podrá interpretar para localizar en el código y averiguar su naturaleza.

Dadas las características asíncronas de las ejecuciones de los kernel en CUDA™ respecto al host cuando se ejecuta el código, se añadirá una función adicional. Será "**cudaDeviceSynchronize()**", su cometido es bloquear la ejecución del código destinado a la GPU hasta que se han terminado todas las tareas anteriores. Es decir, si alguna de las funciones relacionadas con la GPU, por ejemplo reserva o copia de memoria, no ha terminado, la GPU estará parada y no comenzará su trabajo, devolviendo un error si alguna de las tareas ha fallado. Evidentemente esta función afecta al rendimiento del programa, ya que es posible que no sea necesario esperar a todas las funciones anteriores que estén relacionadas con la GPU, ya que podrían estar implicadas diferentes GPUs u otras razones, sin embargo en el caso

estudiado ha sido interesante tanto para conocer su funcionamiento y estudiar errores que sucedieron durante la programación. Además el rendimiento es mínimamente influenciado.

El uso de estas funciones está representado en el proyecto por medio de pequeñas funciones que realizan el control de los errores que puedan surgir:

```
inline void __cudaSafeCall( cudaError err, const char *file, const int line )
{
#ifdef CUDA_ERROR_CHECK
    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n", file, line,
cudaGetErrorString( err ) );
exit( -1 );
    }
#endif
    return;
}
inline void __cudaCheckError( const char *file, const int line )
{
#ifdef CUDA_ERROR_CHECK
    cudaError err = cudaGetLastError();
    if ( cudaSuccess != err )
    {
       fprintf( stderr, "cudaCheckError() failed at %s:%i : %s\n", file, line,
cudaGetErrorString( err ) );
       exit( -1 );
    }
    err = cudaDeviceSynchronize();
    if( cudaSuccess != err )
  {
  fprintf( stderr, "cudaCheckError() with sync failed at %s:%i : %s\n", file,
line, cudaGetErrorString( err ) );
exit( -1 );
```

```
    }
#endif
return;
}
```

Como añadido, pese a no usarse en este proyecto, la función "**cudaDeviceScheduleBlockingSync()**" bloquea la ejecución de código por parte del host hasta que la GPU termine su trabajo. De esta manera se puede conseguir sincronismo entre la GPU y el host, sin embargo esta opción afecta de manera más importante al rendimiento del programa.

4.1.3.4 Liberación de recursos

De manera análoga al lenguaje C original, la memoria reservada deberá ser liberada después de los cálculos pertinentes. Para ello se utilizará la función "**cudaFree()**", de nuevo similar al "**free()**" original de C y que liberará la memoria de la GPU que ha sido utilizada por el programa.

```
cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
```

Adicionalmente se ha usado la función "**cudaDeviceReset()**". Esta función explícitamente destruye y limpia todos los recursos asociados al proceso en curso. La primera llamada a cualquiera de los comandos de la API de CUDA™ tras este función, reseteará la GPU, evitando así errores por elementos que no se hayan eliminado, o posibles residuos de la ejecución. Es recomendable colocar este comando al principio y final del código escrito para asegurar un funcionamiento "limpio" de la GPU.

4.2 Descripción del programa

El código implementado en el programa se trata de una adaptación del algoritmo RELF-3D, creado originalmente en Matlab. La razón de esta adaptación es permitir el uso de la API de CUDA™, para lo cual se ha usado el entorno de desarrollo proporcionado por Microsoft Visual C++ Express 2010, ya que permite la creación de programas usando lenguaje C y las opciones de modificación de librerías son fáciles de usar a la par que amplias. Podrá encontrarse las instrucciones para ello en el apéndice B.

El procedimiento de adaptación del código ha sido:

1) Creación de un entorno tridimensional nuevo.
2) Creación del kernel
3) Adaptación del algoritmo genético.

4.2.1 Creación de un entorno tridimensional nuevo.

En el proyecto original se utilizó un mapeado de la universidad (figura 1.4) utilizando complementos de Matlab para representarlo. En este caso, dada la falta de dicho complemento y que el objetivo es la adaptación del módulo "**dist_est_3D**" del desarrollo

original para acelerar su ejecución, se han creado dos entornos tridimensionales más pequeños.

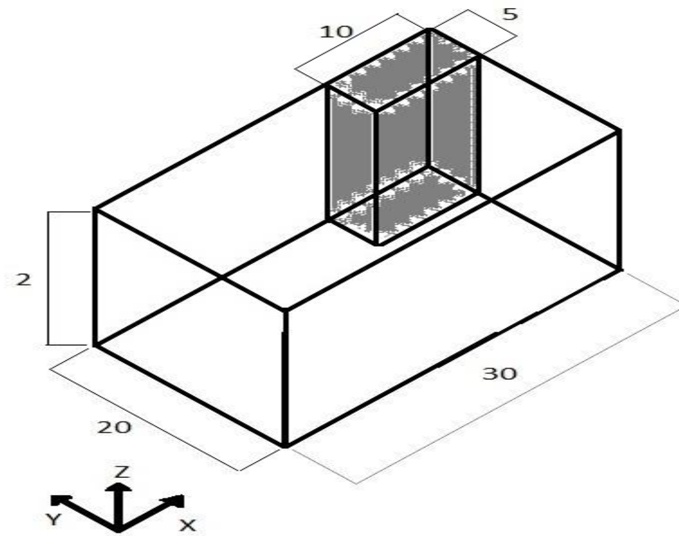

Figura 4.5: Representación entorno tridimensional.

El primer caso se trata de una pequeña habitación vacía, con lo que podría representar una columna en una de sus esquinas, y cuyas medidas pueden observarse en la Figura 4.5.

El segundo caso se trata de un entorno similar, al que han sido añadidas dos paredes de manera que se asemeje a la construcción con pequeños despachos del mapeado utilizado en los experimentos originales. Sus medidas y distribución pueden observarse en la figura 4.6.

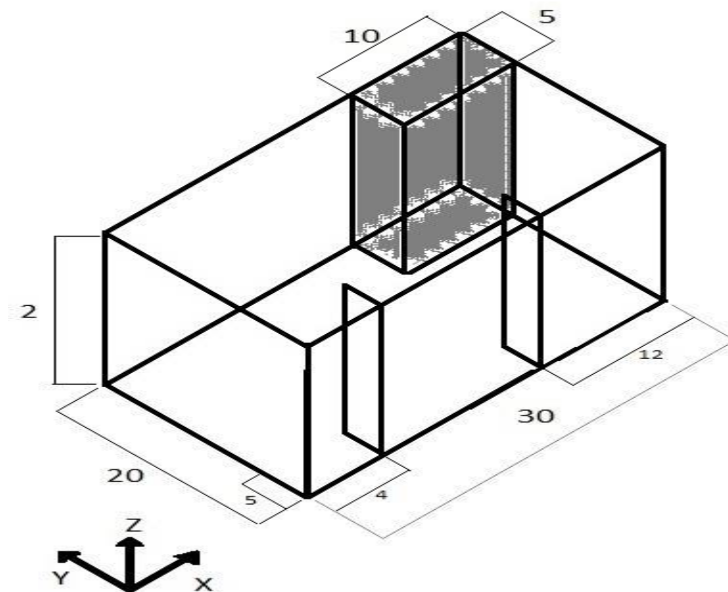

Figura 4.6: Representación entorno tridimensional.

Estos entornos tridimensionales nos permiten utilizar la propia función adaptada "**dist_est_3D**" para usarla una primera vez como si del propio láser del robot MANFRED se tratara. Esto permite crear unos valores iniciales que representan las distancias que mediría el

láser, así como una matriz de coordenadas cartesianas que representan el punto en el que el láser chocaría con una pared. Estas mediciones pueden ser almacenadas en archivos para ser representados como “nubes” de puntos de coordenadas por Matlab, permitiendo así transformar los datos en un equivalente de lo que estaría viendo el robot.

Algunas imágenes de ejemplo:

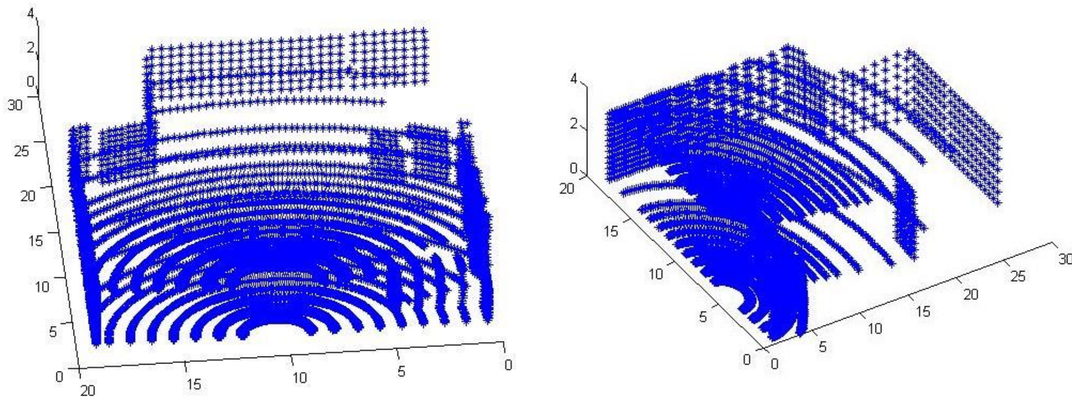

Figura 4.7: Representación nube de puntos masiva.

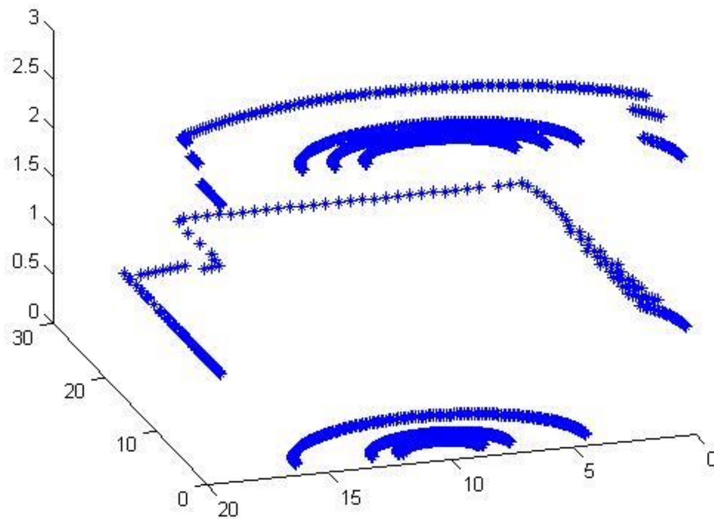

Figura 4.8: Habitación con vista simple.

La figura 4.8 representa uno de los resultados finales que ofrece el programa creado y representaría la vista de la figura 4.5 según la simulación de mediciones por láser. La densidad de puntos es menor que la figura 4.7, obtenida a partir de otro experimento. Esto se debe al número de mediciones que finalmente se ha optado realizar en el presente proyecto.

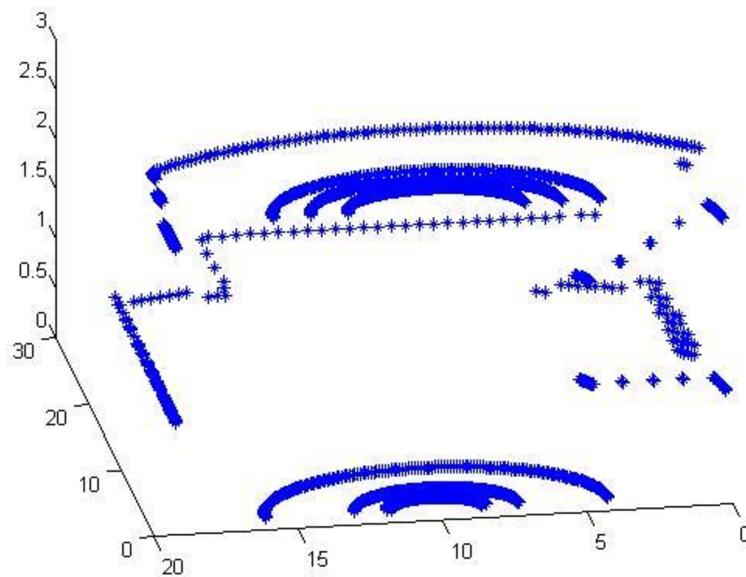

Figura 4.9: Habitación con paredes.

De manera similar, la figura 4.9 es una muestra de una simulación en el segundo entorno creado y mostrado anteriormente en la figura 4.6. Se puede observar la "sombra" que crearían ambas paredes, tras las que las mediciones han llegado, por lo que se confirma que el comportamiento es correcto.

4.2.2 Creación del kernel

El kernel creado adapta el cálculo secuencial de la función "**dist_est_3d**" al cálculo paralelo por CUDA™. Su cometido es estimar las medidas que obtendría el láser desde una posición determinada sin tener que recurrir realmente a este dispositivo, y también para evaluar las distancias de las posiciones de la población generada aleatoriamente. Se utilizará primero para crear la matriz de coordenadas cartesianas y distancias que simularían la realidad desde una posición dada; posteriormente se utiliza para la primera población generada de manera aleatoria, obteniendo el error entre ésta y la real mediante la función **"fitness_3d"**; por último se utiliza en el mecanismo de selección durante la ejecución de la parte del código correspondiente al algoritmo genético. Se pretende así minimizar el tiempo que el algoritmo pasa haciendo estos cálculos.

La manera de afrontar su diseño fue la siguiente (puede ser extrapolada y ser usada con cualquier kernel que se desee crear):

- Reservar la memoria necesaria para copiar las variables que especifican la posición, el ángulo, los máximos y mínimos del mapa, las matrices que almacenarán los resultados, y la matriz que representa el entorno tridimensional.
- Evaluar la cantidad de bloques junto con el número de hilos que contendrán.
- Crear unas variables en las que almacenar el índice de bloque e hilo.
- Definir las ecuaciones de cálculo con las variables de índice de manera que cada hilo represente un ángulo.

Una vez que tenemos las variables definidas en el host, las copiaremos en la GPU usando las funciones ya conocidas de "**cudaMalloc()**" y "**cudaMemcpy()**". Las variables que se copiarán serán la posición, orientación, máximos y mínimos del mapa, el mapa propiamente dicho y las matrices de coordenadas cartesianas y sus distancias. Para reducir el tiempo necesario en copiar estas variables, en las sucesivas llamadas al kernel, sólo se copiarán los datos que se modifiquen y el resto se mantendrán en la GPU. Las variables que se copiarán del host a la GPU y de la GPU al host cuando sean modificadas son posición, orientación y matrices de resultados.

A continuación definimos las variables que almacenarán los índices de los hilos:

```
int tid = threadIdx.x
int bid = blockIdx.x
int tid2 = blockIdx.x * blockDim.x + threadIdx.x
```

Todas ellas son de tipo entero, ya que los índices son siempre números enteros positivos incluyendo el 0. Todas ellas son creadas dentro de cada hilo al ejecutar el kernel. De esta manera cada una de ellas tendrá un valor distinto para cada hilo de manera simultánea pese a tener el mismo nombre. Los hilos no pueden modificar variables de otros hilos, con lo que no supone ningún riesgo. La variable "**tid**" almacena el índice del hilo dentro del bloque en el que se encuentra; la variable "**bid**" almacena el índice de bloque en el que se encuentra; la variable "**tid2**" almacena el número de hilo absoluto en el que se encuentra, teniendo en cuenta todos los anteriores ya que usa el número de bloque, su tamaño y el número de hilo.

Con estos índices ya se tiene capacidad para definir los ángulos que representan la inclinación y giro del láser para tomar medidas: θ y γ.

El ángulo θ representa el giro de derecha a izquierda del ángulo:

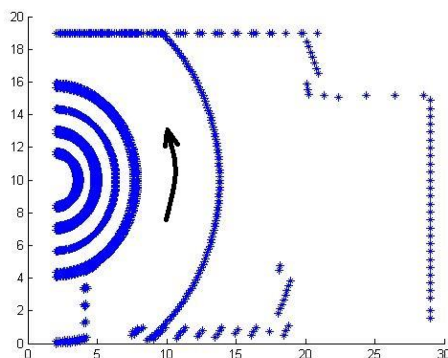

Figura 4.10: Giro horizontal.

Se define: $theta_aux = *theta - (tid * incr_theta)$

Cada ángulo viene definido por el ángulo de la posición que se pasa a la función al ser llamada mediante la variable "theta" y por el número de hilo en el que nos encontremos "**tid**". Los incrementos son de 1°, empezando en 0 hasta 180. Esto hace un total de 181 hilos.

El ángulo γ representa la inclinación del láser:

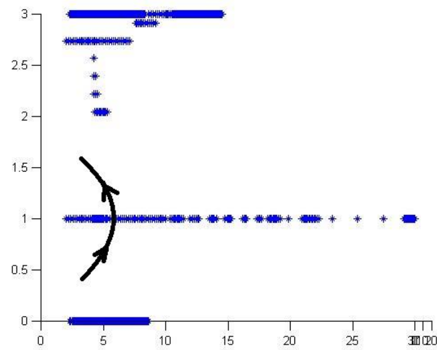

Figura 4.11: Inclinación.

Se define: $phi = -40 + (bid * 10)$

En este caso la inclinación empieza en $-40°$ y asciende hasta $40°$. El incremento se produce según el bloque en el que nos encontremos, siendo un total de 9.

Una vez tenemos los ángulos definidos según el bloque y el hilo, los cálculos de las coordenadas y sus respectivas distancias se realiza mediante trigonometría de coordenadas esféricas.

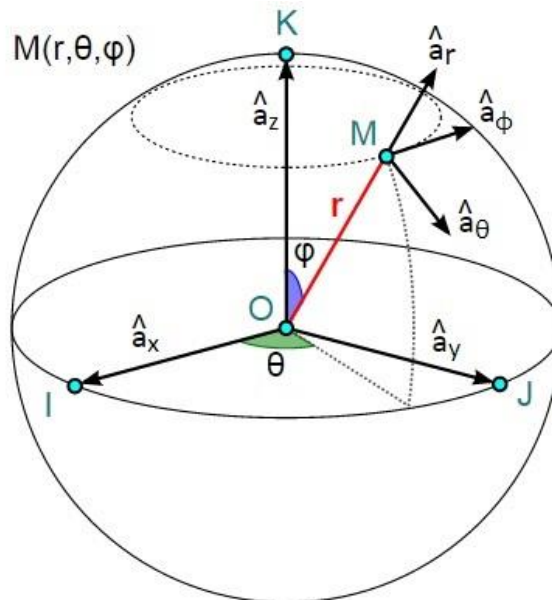

Figura 4.12: Esquema coordenadas esféricas.

Para definir entonces el seno y el coseno del giro horizontal del sensor, teniendo en cuenta que en el lenguaje C las funciones trabajan en radianes, las ecuaciones serán:

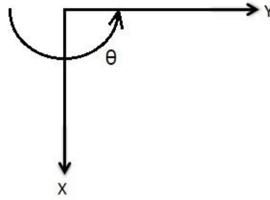

Figura 4.13: Giro horizontal.

```
sin_sensor = sin(((theta_aux - 90) * pi) / 180);

cos_sensor = cos(((theta_aux - 90) * pi) / 180);
```

Se incluye para el ángulo de cálculo de seno y coseno un “-90” para hacer el barrido completo horizontal, ya que el eje de coordenadas ha sido definido de esta manera al crear el entorno tridimensional y es necesario para cubrir los $180°$ de medidas.

Para definir el seno y el coseno del ángulo de inclinación del sensor, las ecuaciones serán:

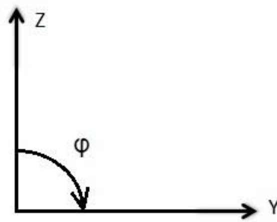

Figura 4.14: Ángulo de inclinación.

```
sin_altura = sin(((90 - phi) * pi) / 180);

cos_altura = cos(((90 - phi) * pi) / 180);
```

Así, para comenzar con inclinación de $-40°$, se incluye el término 90 – phi:

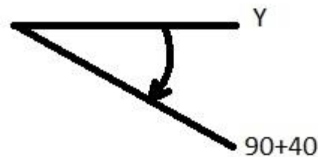

Figura 4.15: Inclinación.

Con esto, se pueden definir los cálculos de las coordenadas X, Y, Z:

$$X = r \cdot sen\varphi \cdot cos\theta$$

$$Y = r \cdot sen\varphi \cdot sen\theta$$

$$Z = r \cdot cos\varphi$$

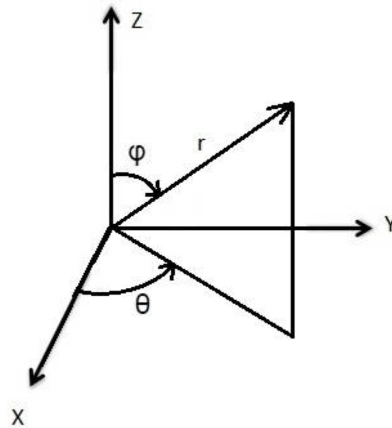

Figura 4.16: Proyección vector.

Con las proyecciones de un vector "**r**" en el sistema de coordenadas definido como en la figura 4.16 podemos obtener los incrementos que utilizaremos para hacer crecer el vector hasta encontrar un punto de la habitación ficticia que represente un obstáculo. Los incrementos serían:

```
incr_x = T * (sin_altura * cos_sensor)

incr_y = T * (sin_altura * sin_sensor)

incr_z = T * cos_altura
```

El factor "**T"** nos permite hacer estos incrementos más grandes o pequeños, de manera que se pueden precisar más las medidas. Se ha tomado un valor base 1, pero puede ser fácilmente modificado.

Los incrementos del vector se realizan dentro de un bucle que no para hasta encontrar obstáculos:

```
while(final == 0){

device_x_round = (int)device_x_posicion;

device_y_round = (int)device_y_posicion;

device_z_round = (int)device_z_posicion;
```

Convertimos la posición a números enteros para poder usarlos en el índice de la matriz de datos. Esto se debe a que en el lenguaje C los índices de una matriz, lógicamente, deben ser variables enteras siempre para evitar errores. Se compara entonces si la posición es un obstáculo.

```
if ( datos[(device_z_round * 30 * 20) + (device_y_round * 30) + device_x_round]
== 1){

final += 1;
```

```
}
```

En caso de no serlo se incrementa el valor de la posición.

```
else{
device_x_posicion = device_x_posicion + incr_x;
device_y_posicion = device_y_posicion + incr_y;
device_z_posicion = device_z_posicion + incr_z;
}
```

Se comparan los valores de la posición con los extremos del mapa para evitar posibles errores y en caso de ser necesario se redondean.

```
if(device_x_posicion > 29){
                                final += 1;
                                device_x_posicion = floor(device_x_posicion);
                                }
                        if(device_y_posicion > 19){
                                final += 1;
                                device_y_posicion = floor(device_y_posicion);
                                }
                        if(device_z_posicion > 3){
                                final += 1;
                                device_z_posicion = floor(device_z_posicion);
                                }
                        if(device_x_posicion < 0){
                                final += 1;
                                device_x_posicion = 0;
                                }
                        if(device_y_posicion < 0){
                                final += 1;
                                device_y_posicion = 0;
                                }
                        if(device_z_posicion < 0){
                                final += 1;
```

```
                        device_z_posicion = 0;

                        }

}
```

A continuación se calcula la distancia. Para ello se calcula la hipotenusa del triángulo formado por las coordenadas finales respecto a las coordenadas originales.

```
dis = sqrt ( pow((device_x_posicion - posicion[0]),2) + pow((device_y_posicion - posicion[1]),2) + pow((device_z_posicion - posicion[2]),2) )
```

Por último se guardan todos los valores de coordenadas y distancias en dos matrices que se devolverán al host para poder ser utilizadas.

```
dist_3d[tid2] = dis

cart_3d[(((bid*blockDim.x) + tid) * 3) + 0] = device_x_posicion

cart_3d[(((bid*blockDim.x) + tid) * 3) + 1] = device_y_posicion

cart_3d[(((bid*blockDim.x) + tid) * 3) + 2] = device_z_posicion
```

Tanto la matriz “**dist_3d**” como la matriz “**cart_3d**” están indexadas usando el número de hilos y/o bloque en el que se encuentran.

4.2.3 Adaptación del algoritmo genético.

La adaptación del algoritmo genético está basada en el algoritmo RELF-3D original, adaptando los límites de bucles y matrices a las características de C, ligeramente diferentes a las de Matlab. Se ha pretendido mantener la máxima fidelidad al algoritmo original para, al realizar las pruebas de comparación de tiempos y rendimiento entre la programación paralela y la programación secuencial, poder tener una objetividad suficiente y centrar las opiniones en si la programación con CUDA es interesante o no. Puede observarse el código completo en el apéndice C.

Capítulo 5: Resultados experimentales

Para apoyar la hipótesis de los beneficios que aportaría el uso de CUDA™ se han realizado ensayos con idénticos algoritmos genéticos y funciones de cálculo de distancias. La diferencia radica únicamente en si se ha programado para ejecutarse secuencialmente o en paralelo. La manera de analizar la diferencia ha sido la ejecución de ambas opciones en los entornos tridimensionales de las figuras 4.5 y 4.6, utilizando coordenadas de posición relevantes para la generación de la simulación de medidas láser. Se han tomado los tiempos de ejecución como diferencia entre ambos, ya que tanto el modo secuencial como el paralelo consiguen la localización de manera satisfactoria y no es un resultado especialmente relevante ni el objetivo del proyecto.

A continuación se muestran las figuras que representan las medidas que tomaría el láser y la simulación de su resultado mediante gráficas de Matlab. Cada figura irá acompañada de la tabla de resultados correspondiente que se utilizarán para calcular la aceleración respecto al original por medio de la "Ley de Amdahl". Este cálculo es en realidad una aproximación de la ganancia de velocidad que ocurre cuando programas secuenciales son modificados para ejecutarse en paralelo. Para ser válida, es necesario que las dimensiones del problema sean las mismas de manera secuencial y paralela. En el caso actual, la adaptación es lo suficientemente aproximada como para poder aplicarse.

Sus parámetros son:

- A = aceleración o ganancia de velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas.
- Am = es el factor de mejora que se ha introducido en el subsistema mejorado.
- Fm = es la fracción de tiempo que el sistema utiliza el subsistema mejorado.

Su formulación es:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \quad (5.1)$$

La interpretación de la ley de Amdahl significa que es el algoritmo el que permite la mejora de la velocidad, no el número de procesadores, ya que al final se llegará a un momento en que no se puede paralelizar más un algoritmo.

5.1 Resultados del entorno tridimensional vacío.

- Posición definida en X=3, Y=3, Z=1, $\theta = 0$

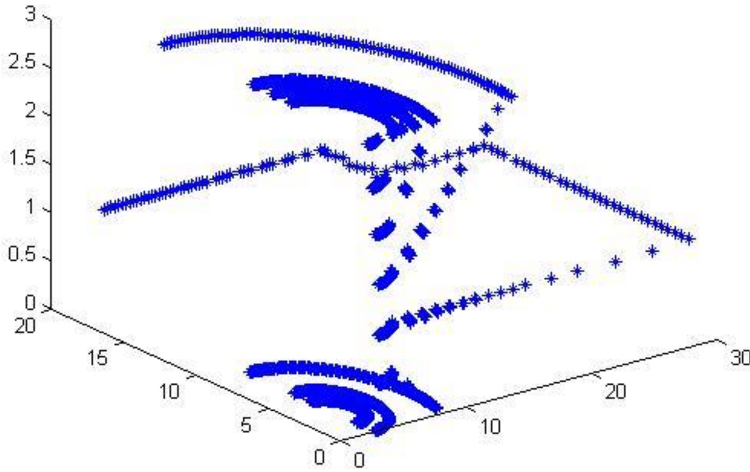

Figura 5.1: Posición (3, 3, 1, 0)

x	3,977298	5,585819	2,905388	3,559941	3,440895
y	3,003793	3,037683	3,002331	3,038855	3,075611
z	0,96268	0,9705	0,994631	0,967503	0,936271
θ	360	359,374878	0,320123	359,48413	0
t CUDA™ [s]	1,295	1,165	1,225	1,15	1,17
t secuencial [s]	20,405	20,412	20,695	20,3	20,555

Tabla 5.1 Resultados posición.

Tiempo medio algoritmo paralelo: 1,201s

Tiempo medio algoritmo secuencial: 20,4734s

$$Media_{paralelo} = 1{,}201\text{s}$$

$$Media_{secuencial} = 20{,}4734\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}9413$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}455$$

- Posición definida en X=10, Y=10, Z=1, $\theta = 0$

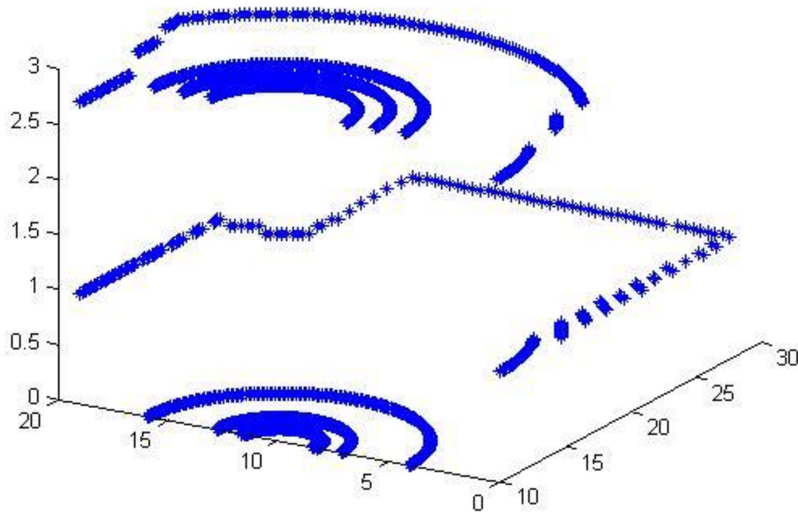

Figura 5.2: Posición (10, 10, 1, 0)

x	10,045041	9,996687	18,938097	9,866026	9,888759
y	10,022587	10,01425	9,923693	10,020892	10,120447
z	0,992931	0,97668	0,986244	0,989678	0,991709
θ	360	360	188,746979	360	359,240784
t CUDA™ [s]	1,155	1,165	1,165	1,165	1,215
t secuencial [s]	20,35	20,79	20,59	20,345	20,9

Tabla 5.2 Resultados posición.

Tiempo medio algoritmo paralelo: 1,173s

Tiempo medio algoritmo secuencial: 20,595s

$$Media_{paralelo} = 1{,}173\text{s}$$

$$Media_{secuencial} = 20{,}595\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}943$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}6$$

- Posición definida en X=15, Y=15, Z=1, $\theta = 0$

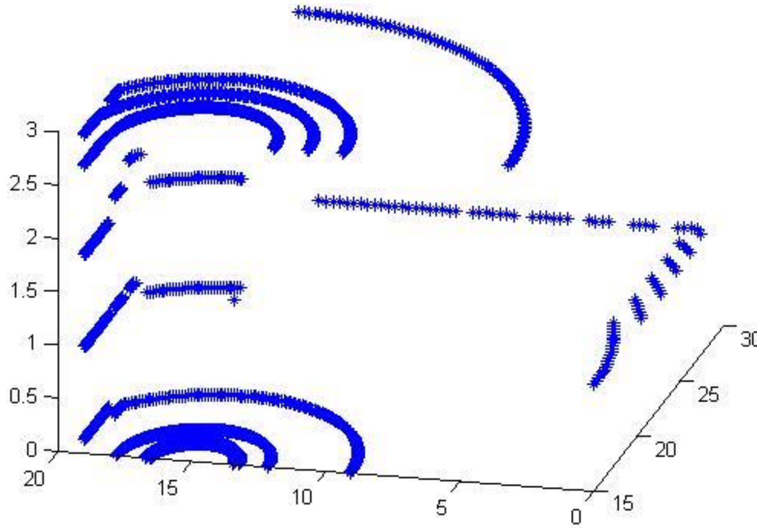

Figura 5.3: Posición (15, 15, 1, 0)

x	14,549724	15,536954	13,674035	15,309161	14,86473
y	14,941854	5,105924	15,002271	5,100479	15,087047
z	0,998483	0,964222	0,987368	0,99435	0,930648
θ	1,585243	208,500488	3,11361	208,111465	0
t CUDA™ [s]	1,23	1,15	1,155	1,195	1,16
t secuencial [s]	20,525	20,31	20,295	20,84	20,63

Tabla 5.3 Resultados posición.

Tiempo medio algoritmo paralelo: 1,178s

Tiempo medio algoritmo secuencial: 20,52s

$$Media_{paralelo} = 1{,}178\text{s}$$

$$Media_{secuencial} = 20{,}52\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}943$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}6$$

- Posición definida en X=22, Y=5, Z=1, $\theta = 0$

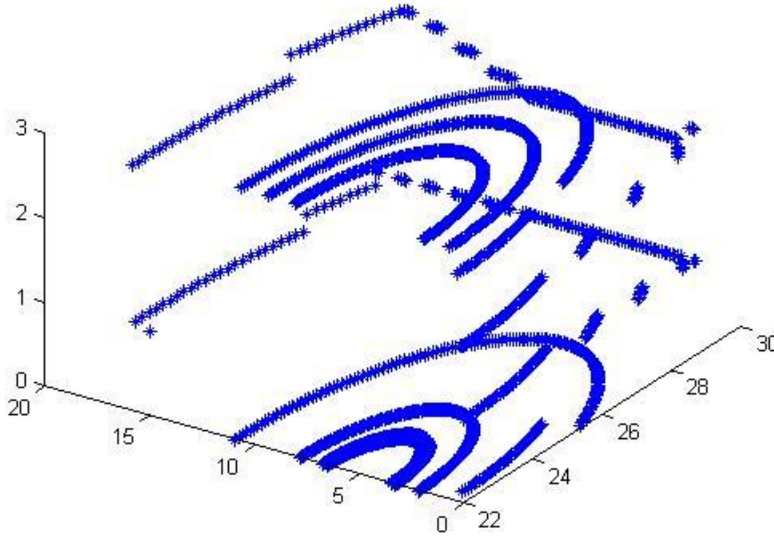

Figura 5.4: Posición (22, 5, 1, 0)

x	21,726967	21,608803	21,501865	7,718497	21,927473
y	5,116076	5,786888	5,579871	14,108172	5,616185
z	0,968922	0,835081	0,997448	0,959363	0,99893
θ	0	0	348,92569	174,986908	357,734833
t CUDA™ [s]	1,25	1,145	1,165	1,17	1,245
t secuencial [s]	20,75	20,335	20,775	20,545	20,33

Tabla 5.4 Resultados posición.

Tiempo medio algoritmo paralelo: 1,195s

Tiempo medio algoritmo secuencial: 20,547s

$$Media_{paralelo} = 1{,}195\text{s}$$

$$Media_{secuencial} = 20{,}547\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}942$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}51$$

5.2 Resultados del entorno tridimensional con obstáculos.

- Posición definida en X=2, Y=2, Z=1, $\theta = 0$

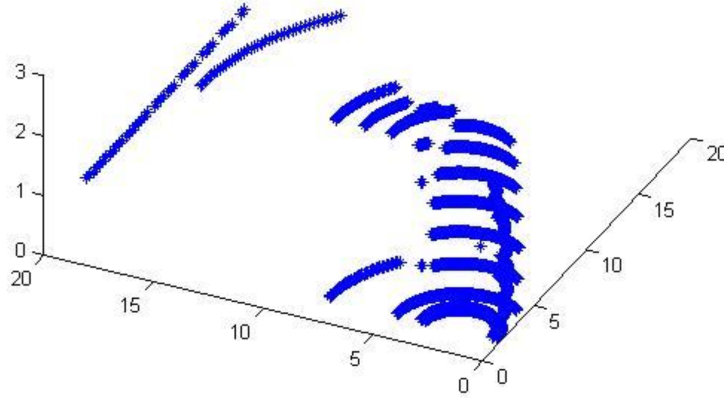

Figura 5.5: Posición (2, 2, 1, 0)

x	1,8537	2,344922	1,878377	2,201105	2,176423
y	2,08789	2,418036	1,973407	2,219105	2,262248
z	0,98262	0,978501	0,85157	1,337969	0,939764
θ	357,98	0	359,375458	3,812744	358,44733
t CUDA [s]	1,046	1,066	1,087	0,995	1,186
t secuencial [s]	20,661	20,099	20,532	20,102	20,613

Tabla 5.5 Resultados posición.

Tiempo medio algoritmo paralelo: 1,076s

Tiempo medio algoritmo secuencial: 20,4014s

$$Media_{paralelo} = 1{,}076s$$

$$Media_{secuencial} = 20{,}4014s$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}947$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}965$$

- Posición definida en X=2, Y=13, Z=1, $\theta = 0$

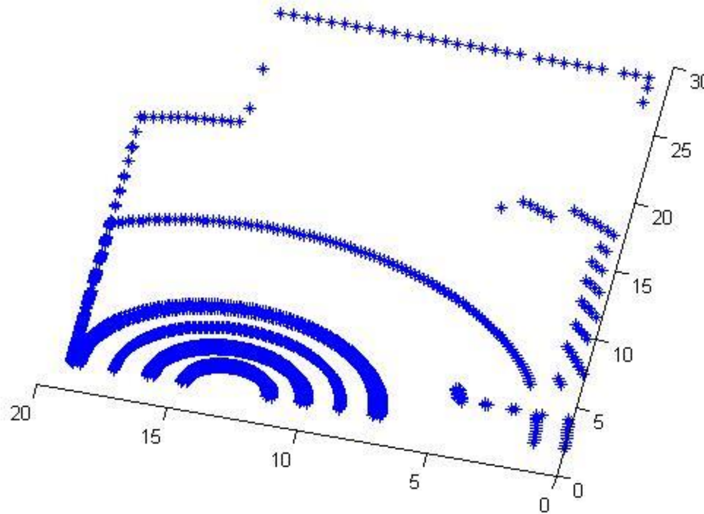

Figura 5.6: Posición (2, 13, 1, 0)

x	1,353906	1,982425	5,205543	14,044112	1,985383
y	12,38045	12,996615	12,460466	5,571609	13,002605
z	0,985789	0,965284	0,987648	0,871532	0,984955
θ	4,342227	0,037922	0	158,954285	0,219481
t CUDA™ [s]	1,189	1,17	1,186	1,126	1,141
t secuencial [s]	20,974	20,495	20,607	20,432	20,656

Tabla 5.6 Resultados posición.

Tiempo medio algoritmo paralelo: 1,1496s

Tiempo medio algoritmo secuencial: 20,6328s

$$Media_{paralelo} = 1{,}1496s$$

$$Media_{secuencial} = 20{,}6328s$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}944$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}6$$

- Posición definida en X=2, Y=18, Z=1, $\theta = 0$

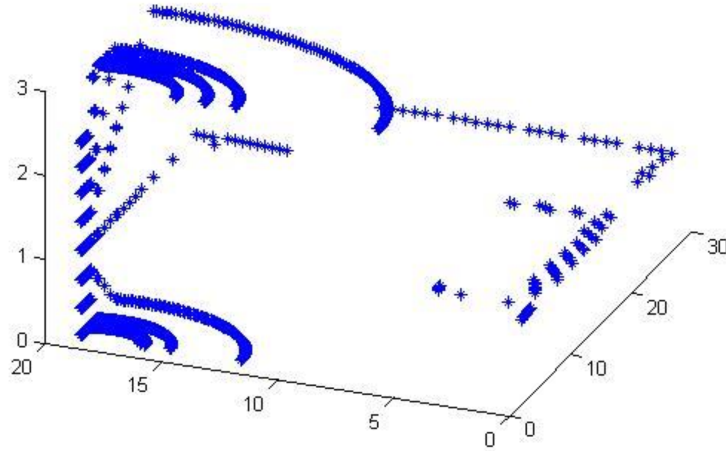

Figura 5.7: Posición (2, 18, 1, 0)

x	28,369041	2,013065	1,843374	2,527415	3,837041
y	1,838399	18,002676	17,99361	17,8804	18,0003
z	0,961445	0,993089	0,996811	0,953741	0,97806
θ	179,590698	0,084394	360	0	0
t CUDA™ [s]	1,309	1,82	1,414	1,161	1,183
t secuencial [s]	20,474	20,597	20,125	20,661	20,609

Tabla 5.7 Resultados posición.

Tiempo medio algoritmo paralelo: 1,3774s

Tiempo medio algoritmo secuencial: 20,4932s

$$Media_{paralelo} = 1{,}3774s$$

$$Media_{secuencial} = 20{,}4932s$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}933$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 8{,}8$$

- Posición definida en X=15, Y=2, Z=1, $\theta = 0$

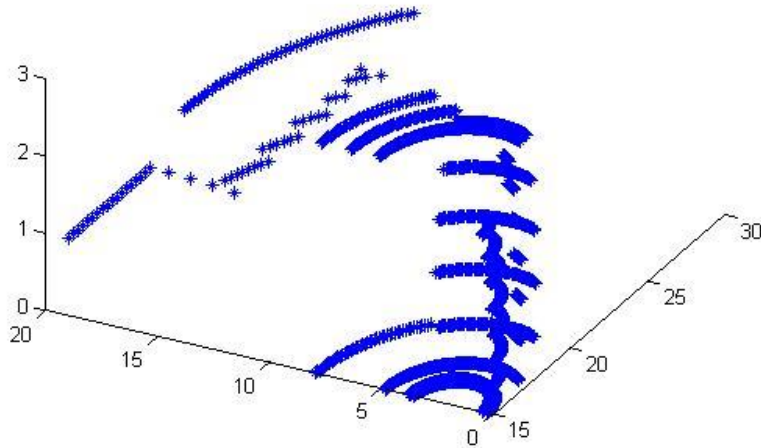

Figura 5.8: Posición (15, 2, 1, 0)

x	18.821.264	13,818766	14,71626	14,733227	15,54245
y	5,491812	2,100719	2,287057	2,403711	2,003681
z	2,023858	0,940131	0,791104	0,866349	0,956786
θ	159,946991	351,1474	352,761169	354,843262	3,252127
t CUDA™ [s]	0,926	1,132	1,135	1,123	1,107
t secuencial [s]	20,357	20,472	20,482	20,428	20,589

Tabla 5.8 Resultados posición.

Tiempo medio algoritmo paralelo: 1,0846s

Tiempo medio algoritmo secuencial: 20,4656s

$$Media_{paralelo} = 1{,}0846\text{s}$$

$$Media_{secuencial} = 20{,}4656\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}947$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}965$$

- Posición definida en X=15, Y=15, Z=1, $\theta = 0$

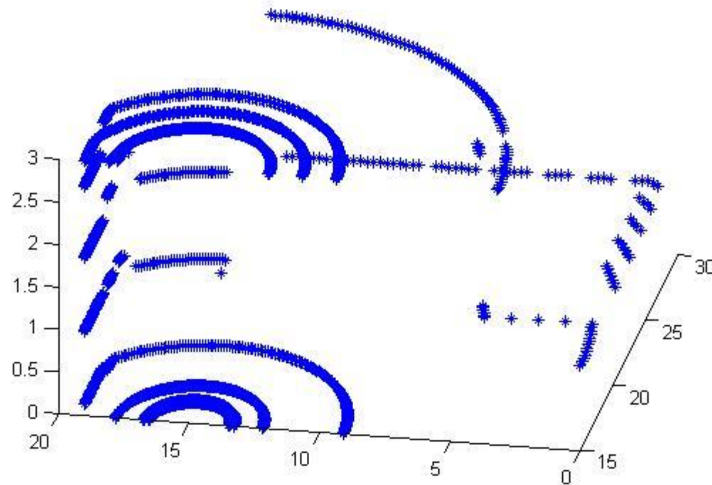

Figura 5.9: Posición (15, 15, 1, 0)

x	14,540052	14,744306	14,611429	15,042869	12,30667
y	14,804836	14,886881	14,801388	15,0818387	4,869746
z	0,988822	0,964982	0,996925	0,948929	0,949676
θ	2,135955	1,385541	2,60715	0	182,738144
t CUDA™ [s]	1,25	1,212	1,14	1,275	1,17
t secuencial [s]	20,47	20,475	20,715	20,15	20,57

Tabla 5.9 Resultados posición.

Tiempo medio algoritmo paralelo: 1,2094s

Tiempo medio algoritmo secuencial: 20,476s

$$Media_{paralelo} = 1{,}2094\text{s}$$

$$Media_{secuencial} = 20{,}476\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}941$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}515$$

5.3 Resultados medios totales

Tiempo total medio algoritmo paralelo: 1,1813s
Tiempo total medio algoritmo secuencial 20,49094s

$$Media_{paralelo} = 1{,}1813\text{s}$$

$$Media_{secuencial} = 20{,}49094\text{s}$$

$$F_m = 1 - \left(\frac{Media_{paralelo}}{Media_{secuencial}}\right) = 0{,}94$$

$$A_m \approx \left(\frac{1}{20}\right)^{-1} \approx 20$$

$$A = \frac{1}{(1 - F_M) + \frac{F_m}{A_m}} = 9{,}35$$

5.4 Experimentos alternativos

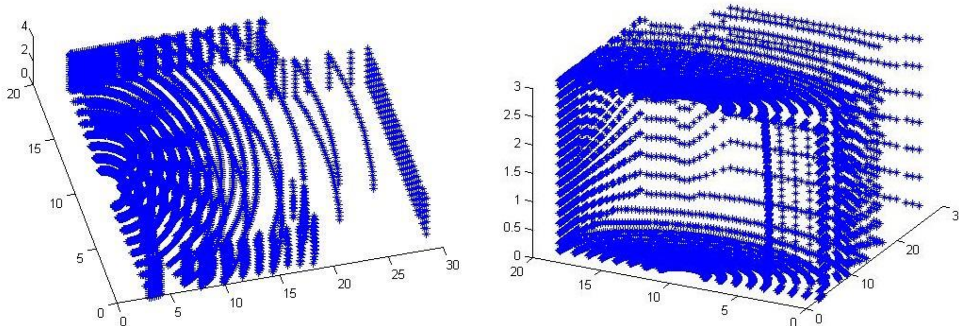

Figura 5.10: Posición (3, 10, 1, 0)

X	4,55768	1,991928	7,162939	14,264881	6,247025
Y	8,776837	5,334024	2,805651	6,138475	13,297364
Z	1	1	1	1	1
Θ	197,890213	18,098356	246,379959	8,77763	0
t CUDA™ [s]	5,396	5,578	5,469	5,403	5,694

Tabla 5.10 Resultados posición.

Como muestra la figura y su correspondiente tabla de resultados, se ha multiplicado por diez la cantidad de puntos que calculará la simulación de medidas del láser, de manera que se ha pasado de 1629 a 16290 medidas. Puede observarse que el tiempo de ejecución ha subido hasta 5 segundos. Se ha comprobado con esto que efectivamente la ley de Amdahl se cumple, al ser efectivamente el diseño del algoritmo el que dicta el tiempo de ejecución y su capacidad de actuar en paralelo, no el número de hilos de ejecución. Por tanto la parte en la que se

ejecuta el algoritmo genético es ahora la que consume más tiempo, por lo que el siguiente paso de estudio será su adaptabilidad a un entorno paralelo.

5.5 Comentarios acerca de los resultados.

Observando primero los resultados obtenidos en el cálculo de posición es fácil identificar lo comentado en el punto 1.1 del capítulo 1. Dependiendo del lugar en el que se encuentre el punto inicial, el algoritmo puede encontrar similitudes con otro punto e identificar su posición de manera "errónea", ya que si, por ejemplo, se encuentra en una esquina, el algoritmo podría encontrar una población que coincida en otra esquina debido a la posición y ángulo. Evidentemente esto no es un fallo de comportamiento en el algoritmo, sino una coincidencia. De todos modos, se puede observar en las tablas adjuntadas que la localización del punto se realiza de manera correcta y precisa.

En términos de tiempo se observa en todas las ejecuciones que el algoritmo con la función paralela tiene un tiempo de solución de sólo unas décimas por encima del segundo y, sin embargo, el algoritmo de ejecución secuencial tiene aproximadamente veinte segundos de tiempo de ejecución, siendo esto una diferencia apreciable.

En cuanto al cálculo de la ley de Amdahl, se puede observar en todas las mediciones que el resultado es de aproximadamente nueve. Esto significa que, siendo un resultado de 1 para la ley de Amdahl una ganancia de velocidad equivalente al 0%, la aceleración obtenida por la presente adaptación equivale a un 460%.

Capítulo 6: Conclusiones y trabajo futuros

6.1 Consecución de objetivos

El objetivo planteado era paralelizar el algoritmo para comprobar si se obtendría una mejora de rendimiento con ello. Es evidente que no sólo se ha conseguido, si no que la mejora obtenida es muy sustancial, por lo que es muy atractivo su uso para el algoritmo RELF3D. Ello permitirá al robot MANFRED realizar tareas de manera mucho más rápida y, en términos de mejora del algoritmo, dado que la parte de mayor cálculo sería más liviana, centrar los esfuerzos en el desarrollo futuro de los métodos de mutación, cruce, selección y descarte.

Queda demostrado también la rentabilidad de uso de CUDA™, ya que no exige, en términos de aprendizaje, más que el dominio de cálculo matricial y la gestión de memoria para la transmisión de datos entre GPU y host, a parte de las nociones básicas de programación.

Además, la programación mediante funciones kernel implica que es de fácil modificación, ya que los datos son variables que se definen al ejecutar la función, siendo por tanto fácilmente portable o editable. Un ejemplo de edición del kernel se muestra en la figura 5.10 en relación a los experimentos alternativos. Es a partir de este experimento que podemos plantear líneas de investigación futuras cuyo objetivo sean posibles mejoras del algoritmo, reestructuración del código en busca de una posible mejora de rendimiento o el estudio de un balance entre el número de mediciones simuladas y el tiempo de ejecución que puedan ser óptimos en términos de rendimiento.

6.2 Ventajas de desarrollo en CUDA™

Como se ha demostrado, la programación mediante CUDA™ tiene un gran potencial por su capacidad de cálculo, su innovación y su facilidad de desarrollo. La facilidad de desarrollo viene además apoyada por la arquitectura propia de CUDA™, ya que permite al sistema escalar en términos de capacidad, sin cambios a nivel de programa, con sólo cambiar de GPU por modelos superiores al ser siempre compatibles.

6.3 Desarrollo portable.

Es evidente que un problema que se podría plantear es la capacidad de implantarse las GPGPU en sistemas robóticos móviles, dado que en su gran mayoría tienen unas medidas que condicionarían el diseño y construcción de éstos. Actualmente tampoco supone un problema, ya que en el mercado también existe un producto recientemente lanzado por NVIDIA® llamado Jetson TK1.

Se trata de una placa base de pequeñas dimensiones preparada para ser integrada en un sistema de espacio limitado y que incluye el chip Tegra®K1 de NVIDIA®, basado en la arquitectura Kepler que actualmente tienen muchas GPU del mercado con capacidad para desarrollo en la plataforma CUDA™, junto con numerosas compatibilidades y características adicionales.

Figura 6.1: NVIDIA® Jetson TK1.

Apéndice A: Costes del proyecto

En este apéndice se realiza una estimación del coste equivalente del proyecto. De manera aproximada se realizará una estimación del tiempo invertido para cada etapa del mismo.

En la tabla A.1 se estimarán los periodos de cada fase del proyecto, mientras que en la tabla A.2 se estimarán los costes relacionados al tiempo estimado, así como de los materiales necesitados.

Fases	***Tiempo(h)***
Estudio y comprensión del problema	10
Iniciación a C y CUDA™	10
Implementación del código	80
Pruebas e identificación de errores	30
Corrección de errores	20
Optimización del código	20
Adaptación respecto a Matlab	10
Corrección de errores	10
Realización de memoria del proyecto (de manera simultánea a las fases anteriores)	120
Total	310

Tabla A.1 Inversión temporal.

Para el cálculo del coste horario, tomaremos como salario aproximado de ingeniero 20€/hora.

Concepto	***Coste***
Coste asociado a la inversión temporal	310h · 20€/h = 6200€
Materiales: Pc con GPU compatible con CUDA™	700€
Software utilizado: Microsoft Visual C++ Express, NVIDIA® CUDA SDK. (Descargables gratuitos).	0€
Total	6900€

Tabla A.2 Coste total asociado al proyecto.

No se incluye el I.V.A. en el cálculo de los costes del proyecto.

Apéndice B. Manual del usuario

Para adaptar el compilador a las librerías de CUDA™ se deben seguir estos pasos:

1) Bajar e instalar en el siguiente orden:
   - Los controladores recomendado por el fabricante para el sistema operativo en el que se desarrollará el programa.
   - El kit de desarrollo proporcionado por NVIDIA®.
   - La actualización del kit de desarrollo que corrige sus "bugs".
   - Kit de desarrollo para computación en GPU.

   Pueden encontrarse todos en: https://developer.nvidia.com/cuda-downloads

2) Comprobar la instalación.
   Para ello abrir el símbolo del sistema y usar el comando "nvcc –V". Con ello comprobaremos que se ha instalado correctamente el controlador de compilación.

3) Comprobación usando un programa de C/C++ pre compilado para CUDA™.
   Este paso es para ver si el dispositivo GPU puede ser usado por programas CUDA C/C++. Para ello localizaremos el "bandwidthTest" que viene incluido en el kit de desarrollo. Un ejemplo de localización de este programa es la carpeta en la que se instala por defecto: "C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0\C\bin\win64\Release". Se ejecutará en símbolo del sistema y comprobaremos si efectivamente el funcionamiento es correcto.

4) Configuración del entorno de desarrollo para CUDA™.
   - Comenzaremos creando un proyecto nuevo.

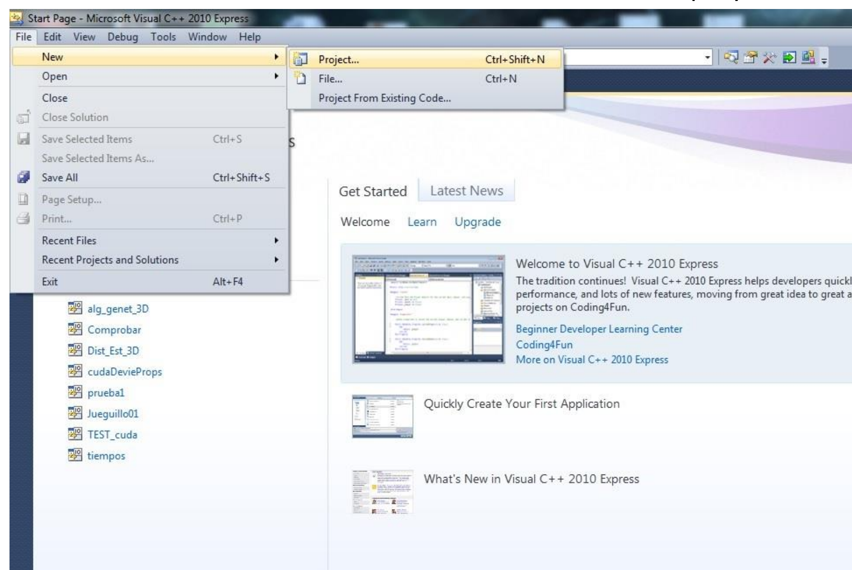

Figura B.1: Crear nuevo proyecto.

- Elegimos aplicación para consola en modo Win32.

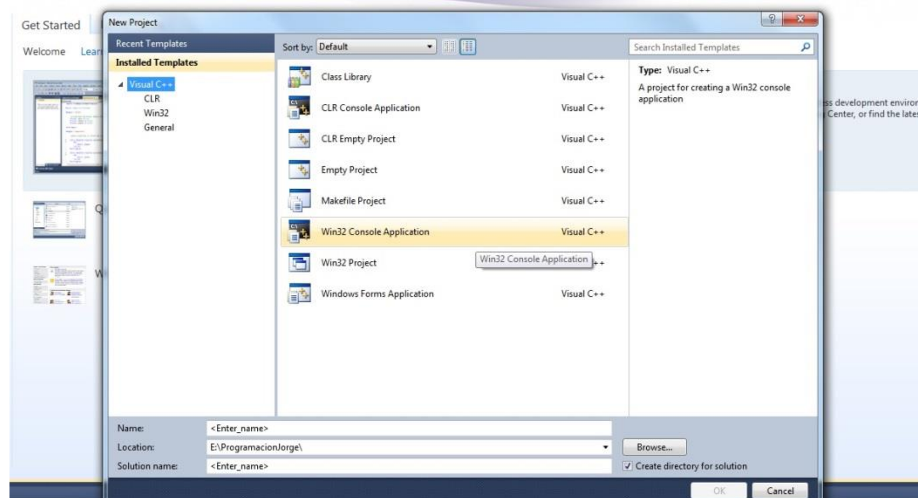

Figura B.2: Opción Win32.

- Nombramos el proyecto

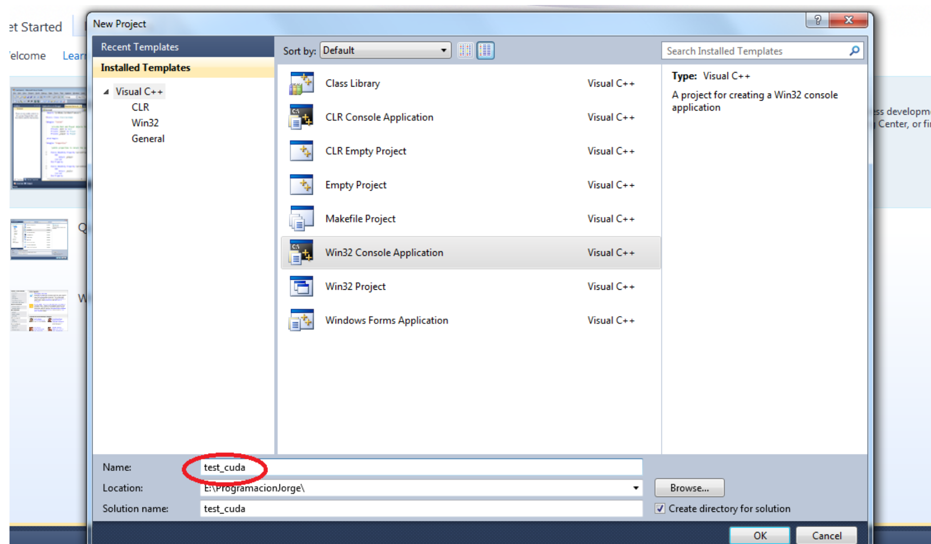

Figura B.3: Nombrar.

- A continuación elegimos aplicación de consola y proyecto vacío.

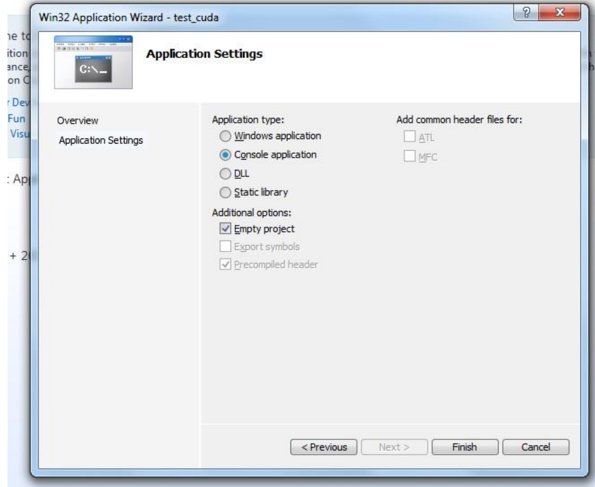

Figura B.4: Opciones de creación de proyecto.

- Finalizaremos la creación del proyecto y podremos entrar en el entorno de desarrollo. Con el botón derecho del ratón en el nombre del proyecto entraremos en el menú de personalización.

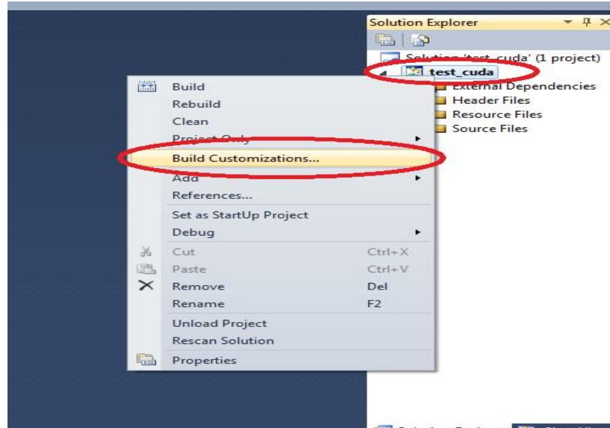

Figura B.5: Opciones de personalización.

- Seleccionaremos la opción de CUDA™

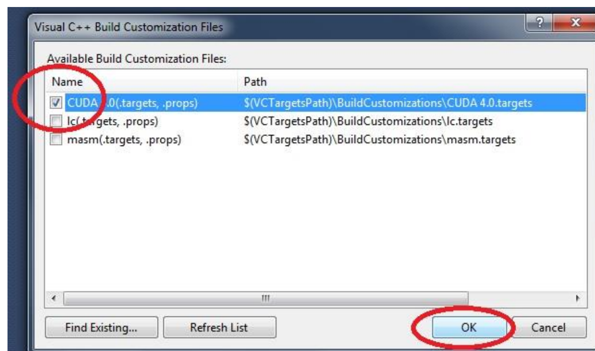

Figura B.6: Personalización para CUDA™.

- De nuevo usando el botón derecho del ratón desplegaremos el menú alternativo y entraremos en la opción de propiedades.

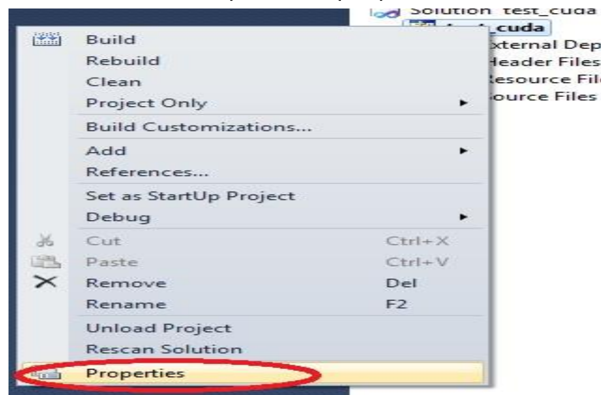

Figura B.7: Propiedades.

- A continuación adjuntaremos la librerías de CUDA™

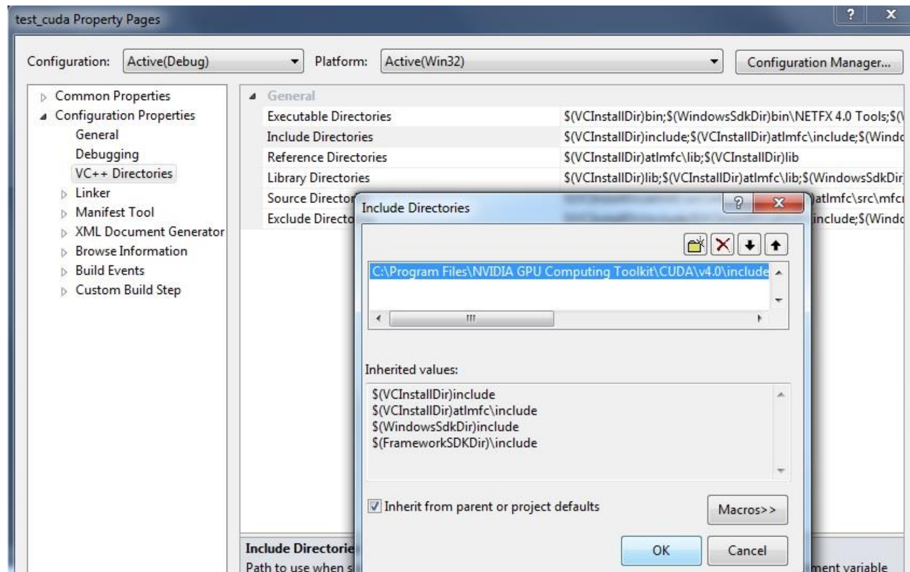

Figura B.8: Inclusión de directorios.

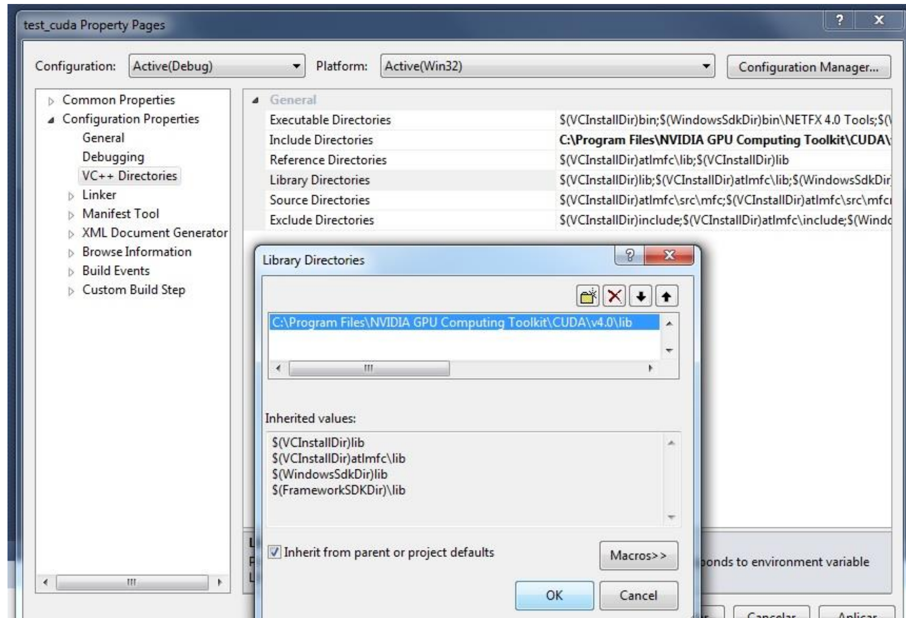

Figura B.9: Inclusión de librerías.

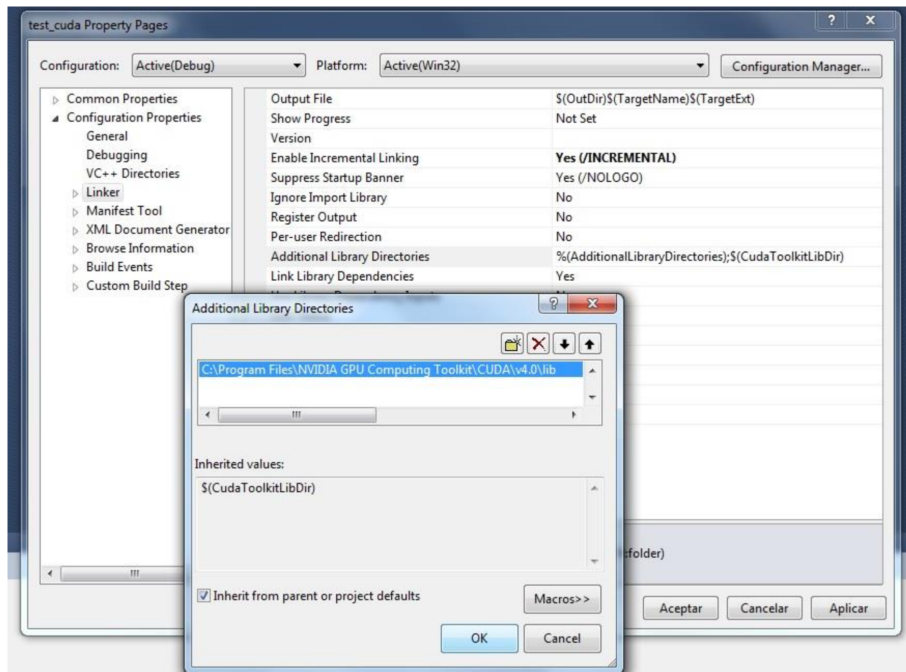

Figura B.10: Librerías adicionales en menú Linker.

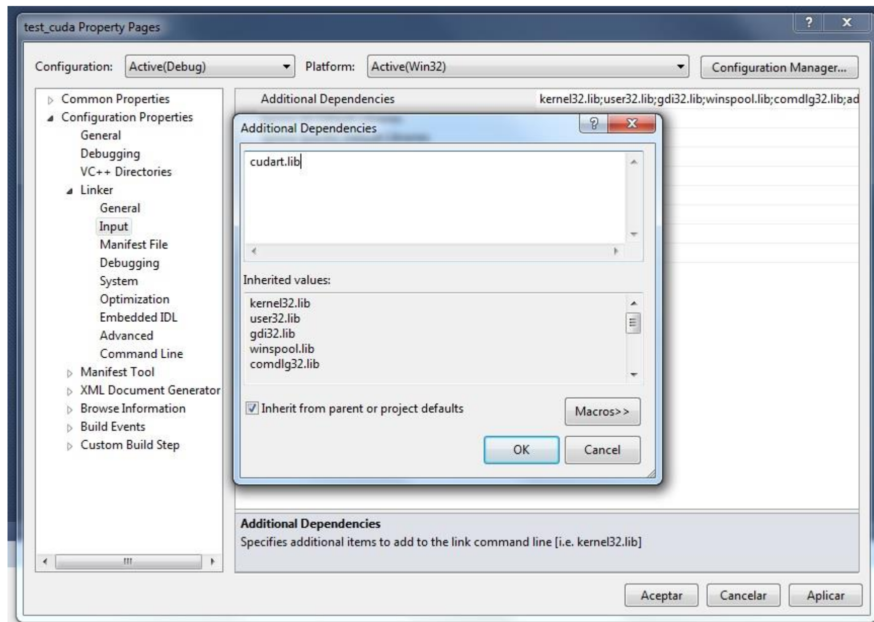

Figura B.11: Dependencias adicionales.

5) Crear código fuente de CUDA™ en C/C++.
   Añadiremos la extensión "**.cu**" al nombre del programa para que el compilador identifique que se trata de código fuente hecho con librerías de CUDA™.

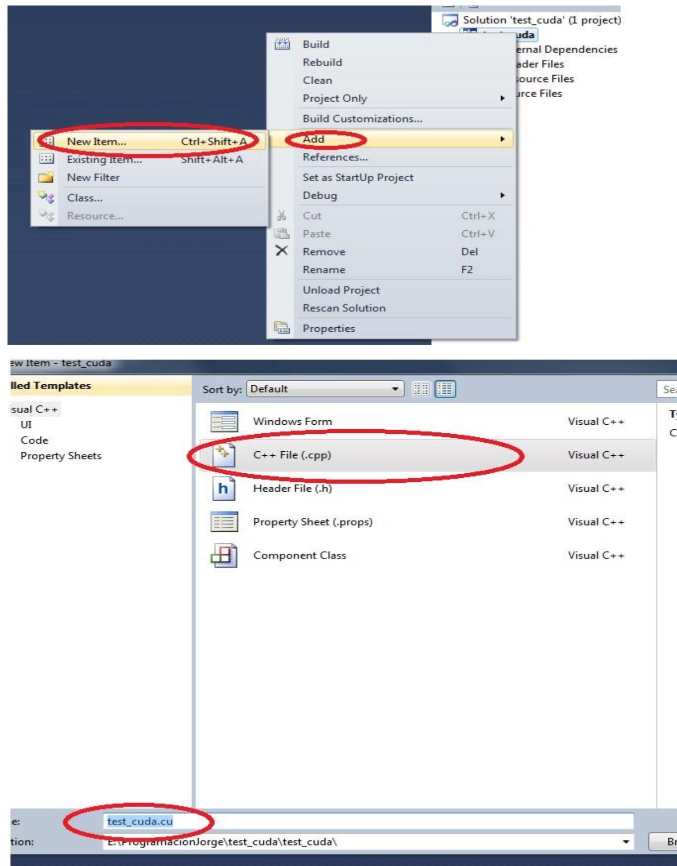

Figura B.12: Nombrar con extensión CUDA™

Tras estos pasos, el compilador ya identificará correctamente el programa al compilar y usará las librerías de CUDA™, con lo que podremos crear el programa sin complicaciones.

Apéndice C. Código del proyecto

```
#include <stdio.h>

#include <math.h>

#include <iostream>

#include <cuda.h>

#include <time.h>

#include <stdlib.h>


//variables que usaremos que no variarán ----------------------------------------
--------------------------------------------------------------------------------
--------------------

// COMENTAR SI SIEMPRE SON IGUAL O INTERESA PODER VARIARLAS

#define pi 3.14159265358979323846

#define num_medidas 181

#define num_barridos 9

#define T 1

#define incr_theta 1

#define incr_phi 5

#define cell_size 0.121

#define D 4 //numero de cromosomas

#define err_dis 0.1

#define CR 0.75


// Funciones para manejo de excepciones


#define CudaSafeCall( err ) __cudaSafeCall( err, __FILE__, __LINE__ )

#define CudaCheckError()    __cudaCheckError( __FILE__, __LINE__ )
```

```
//--------------------KERNEL------------------------------------------------------------------------------------------------------------------------------------------------------

__global__ void kernel( float *theta, float *mapmin, float *mapmax, float *posicion, float *cart_3d, float *dist_3d, int *datos ) {

//-----VARIABLES DENTRO DEL KERNEL-------------------

int final = 0;

double dis = 0;

//float cart[3] = {0, 0, 0};

//VARIABLES TRIGONOMETRICAS

float sin_sensor = 0;

float cos_sensor = 0;

float sin_altura = 0;

float cos_altura = 0;

//INCREMENTOS

float incr_x = 0;

float incr_y = 0;

float incr_z = 0;

//VARIABLES DE REDONDEO DE POSICION Y SUS AUXILIARES

int device_x_round;

int device_y_round;
```

```
int device_z_round;


float device_x_posicion = posicion[0];

float device_y_posicion = posicion[1];

float device_z_posicion = posicion[2];




//VARIABLES PARA INDEXAR BLOQUES E HILOS (BLOCKS & THREADS)


int tid = threadIdx.x;

int bid = blockIdx.x;

int tid2 = blockIdx.x * blockDim.x + threadIdx.x;


//ANGULO PHI INDEXADO SEGUN EL BLOQUE

//phi lleva -40 por ser ángulo inicial predefinido. PHI= angulo inicial + blockIdx*incremento

int phi = -40 + (bid * 10);




// THETA INDEXADO SEGUN EL HILO

int theta_aux = *theta - (tid * incr_theta);


//COMPROBACION DE HILO MENOR QUE EL Nº DE MEDIDAS: O --> NUM_MEDIDAS-1


if (tid2 < num_medidas * num_medidas) {

//-------ASIGNACION DE VALORES--------------------

sin_sensor = sin(((theta_aux - 90) * pi) / 180);

cos_sensor = cos(((theta_aux - 90) * pi) / 180);

sin_altura = sin(((90 - phi) * pi) / 180);

cos_altura = cos(((90 - phi) * pi) / 180);
```

```
//-------INCREMENTOS------------

incr_x = sin_altura * cos_sensor;

incr_y = sin_altura * sin_sensor;

incr_z = cos_altura;

if ( (incr_x * incr_x )< 0.0001) { incr_x = 0;}

if ( (incr_y * incr_y )< 0.0001) { incr_y = 0;}

if ( (incr_z * incr_z )< 0.0001) { incr_z = 0;}


//SINCRONIZAR------------

__syncthreads();

//----------------------


//---------BUCLES--------------------------




while(final == 0){ //Bucle que no para hasta que encuentra 2 celdas con obstáculos

device_x_round = (int)device_x_posicion;

device_y_round = (int)device_y_posicion;

device_z_round = (int)device_z_posicion;


if ( datos[(device_z_round * 30 * 20) + (device_y_round * 30) + device_x_round] == 1){

final += 1;


}//( datos[(device_z_round * 30 * 20) + (device_y_round * 30) + device_x_round] == 1)

else{

device_x_posicion = device_x_posicion + incr_x;

device_y_posicion = device_y_posicion + incr_y;

device_z_posicion = device_z_posicion + incr_z;

}//else
```

```
if(device_x_posicion > 29){
                        final += 1;
                        device_x_posicion = floor(device_x_posicion);
                        }
                if(device_y_posicion > 19){
                        final += 1;
                        device_y_posicion = floor(device_y_posicion);
                        }
                if(device_z_posicion > 3){
                        final += 1;
                        device_z_posicion = floor(device_z_posicion);
                        }

                if(device_x_posicion < 0){
                        final += 1;
                        device_x_posicion = 0;
                        }
                if(device_y_posicion < 0){
                        final += 1;
                        device_y_posicion = 0;
                        }
                if(device_z_posicion < 0){
                        final += 1;
                        device_z_posicion = 0;
                        }

__syncthreads();
}//while(final == 0)

//DISTANCIA
```

```
dis = sqrt ( pow((device_x_posicion - posicion[0]),2) + pow((device_y_posicion -
posicion[1]),2) + pow((device_z_posicion - posicion[2]),2) );



//SINCRONIZAR------------

__syncthreads();

//-----------------------


dist_3d[tid2] = dis;


cart_3d[(((bid*blockDim.x) + tid) * 3) + 0] = device_x_posicion ;

cart_3d[(((bid*blockDim.x) + tid) * 3) + 1] = device_y_posicion ;

cart_3d[(((bid*blockDim.x) + tid) * 3) + 2] = device_z_posicion ;





//SINCRONIZAR------------

__syncthreads();

//-----------------------


}//if (tid < num_medidas)

__syncthreads();

}//__global__ void kernel


//-------------------------------------------

//FIN DEL KERNEL------------------------------

//-------------------------------------------


//****************************************************************************
*************************

//FITNESS_3D!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
//*****************************************************************************
**************************


float fitness_3d(float *estimada_3d, float *real_3d) { //Abre la funcion

       float suma1[num_medidas];

       for(int i = 0; i<num_medidas; i++){

        suma1[i] = 0;

       }

       int otroContador = 0;

       float suma_total = 0;

       for(int j = 0; j<num_medidas; j++){

              for(int i = 0; i < 9; i++){

              suma1[j]     =     suma1[j]     +     (((real_3d[otroContador]     -
estimada_3d[otroContador]) * (real_3d[otroContador] - estimada_3d[otroContador]))
/   (1   +   (2   *   ((err_dis   *   real_3d[otroContador])   *   (err_dis   *
real_3d[otroContador])))));

              otroContador ++;

              }

       }


  for(int i = 0; i<num_medidas; i++){

        suma_total = suma_total + suma1[i];


 }

       return suma_total;


}//Cierra la funcion






//-----------------------------------------------------
```

```
//ERRORES

//---------------------------------------------------

inline void __cudaSafeCall( cudaError err, const char *file, const int line )

{

#ifdef CUDA_ERROR_CHECK

    if ( cudaSuccess != err )

    {

        fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n",

                 file, line, cudaGetErrorString( err ) );

//MIO

getchar();

        //

exit( -1 );

    }

#endif


    return;

}


inline void __cudaCheckError( const char *file, const int line )

{

#ifdef CUDA_ERROR_CHECK

    cudaError err = cudaGetLastError();

    if ( cudaSuccess != err )

    {

        fprintf( stderr, "cudaCheckError() failed at %s:%i : %s\n", file, line,
cudaGetErrorString( err ) );


              getchar();


        exit( -1 );
```

```
    }


    err = cudaDeviceSynchronize();

    if( cudaSuccess != err )

    {

        fprintf( stderr, "cudaCheckError() with sync failed at %s:%i : %s\n",
file, line, cudaGetErrorString( err ) );


              getchar();


        exit( -1 );

    }

#endif


    return;

}

//-------------------------------------------------------

//FIN DE ERRORES

//-------------------------------------------------------


//----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------------------

//----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------------------

//       mmmmmm           mmmmmm           mmmmmmm          mmmm     mmmm    mmmm

//       mmmmmmm         mmmmmmm          mmmm  mmmm        mmmm     mmmmm   mmmm

//       mmmm  mmm       mmm  mmmm       mmmm    mmmm       mmmm     mmmmmm  mmmm

//       mmmm   mmm     mmm   mmmm      mmmm      mmmm      mmmm     mmmmmmmmmmm

//       mmmm    mmm   mmm    mmmm     mmmmmmmmmmmmmmmm     mmmm     mmmm  mmmmmm

//       mmmm     mmm mmm     mmmm    mmmm          mmmm    mmmm     mmmm   mmmmm

//       mmmm      mmmmm      mmmm   mmmm            mmmm   mmmm     mmmm    mmmm
```

```
//--------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------------------

//--------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------------------

int main( void ) {

cudaDeviceReset ();

clock_t start = clock();




float posicion[4] = {15, 15, 1, 0}; // VALORES DE POSICIÓN INICIAL

float *dev_posicion;


int host_x_round = (int) posicion[0];

int host_y_round = (int) posicion[1];

int host_z_round = (int) posicion[2];



float theta = (posicion[3] + 180);

float *dev_theta;



int ERROR01 = 0;






//------------------------------------------->   CREACIÓN  ENTORNO  TRIDIMENSIONAL
<------------------------------------


int m;
```

```
int n;
int o;
int Mapa_3D[30][20][4];
for (m=0; m<30; m++ ){
for (n=0; n<20; n++ ){
for (o=0; o<4; o++ ){
// INICIALIZO LA MATRIZ ENTERA A 0.
Mapa_3D[m][n][o] = 0;
}
}
}

//SUELO: Rellenamos el suelo con 1
for (m=0; m<30;m++){
for (n=0;n<20; n++){
Mapa_3D[m][n][0] = 0;
}
}



//TECHO: Rellenamos el techo con 1
//for (m=0; m<30;m++){
//     for (n=0;n<20; n++){
//     Mapa_3D[m][n][3] = 1;
//     }
//}

//PARED A
for (m=0; m<20; m++){
for (o=0; o<4; o++){
Mapa_3D[m][19][o] = 1;
```

```
}
}

//PARED B
for (n=0; n<20; n++){
for (o=0; o<4; o++){
Mapa_3D[0][n][o] = 1;
}
}

//PARED C
for (m=0; m<30; m++){
for (o=0; o<4; o++){
Mapa_3D[m][0][o] = 1;
}
}

//PARED D
for (n=0; n<15; n++){
for (o=0; o<4; o++){
Mapa_3D[29][n][o] = 1;
}
}

//PARED E-F
for (m=20; m<30; m++){
for (n=15; n<20; n++){
for (o=0; o<4; o++){
Mapa_3D[m][n][o] = 1;
}
}
```

```
}

//AQUI SE CREAN LOS OBSTACULOS. SI SE QUITA CREA LA HABITACIÓN VACÍA
for(int j=0; j < 5; j++){
	for(int k =0; k<4; k++){
		Mapa_3D[4][j][k] = 1;
		}
	}

	for(int j=0; j < 5; j++){
	for(int k =0; k<4; k++){
		Mapa_3D[18][j][k] = 1;
		}
	}
//fin OBSTACULOS

//Copiamos Mapa_3D a una aux lineal
int datos[2400];
int *dev_datos;
int indice=0;
for(int zz=0;zz<4;zz++){
	for(int yy=0;yy<20;yy++){
		for(int xx=0;xx<30;xx++){
			datos[indice] = Mapa_3D[xx][yy][zz];
			indice++;
		}
	}
}
```

```
float mapmax [4] = {m, n, o, 360};

float *dev_mapmax;



float mapmin [3] = {0, 0, 0};

float *dev_mapmin;




//*********************************************************************************
*********************************************************************************
**********************************************

//PPPPPPP        OOOO        BBBBBBBB     LLL

//PPPPPPPP     OOOOOOOO      BB    BBB    LLL

//PP   PPP    OOO    OOO     BB    BBB    LLL

//PP   PP     OOO    OOO     BB   BBB     LLL

//PPPPP       OOO    OOO     BBBBBB       LLL

//PP          OOO    OOO     BB   BBB     LLL

//PP          OOO    OOO     BB    BBB    LLL

//PP           OOOOOOOO      BB    BBB    LLLLLLLLLLL

//PP             OOOO        BBBBBBBB     LLLLLLLLLLL

//---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------------------------

float NP;

int NP_aux;

NP = ceil(sqrt(mapmax[0]*mapmax[1]));

NP_aux = (int)NP;

int NP_round = NP_aux;



//numero random
```

```
float azar;

 srand (time(NULL));

 float *poppre;
 poppre = (float*) malloc((NP_round * D) * sizeof(float));
 float *cost;
 cost =(float*) malloc((NP_round * 1) * sizeof(float));
 int contador = 0;
 int contador2 = 0;
 for (int i = 0; i < NP_round; i++){//for
        cost [contador] = 0;
     for (int j = 0; j < D; j++){//for
         azar = (rand() % 100 + 1) * (rand() % 100 + 1);
         azar = (sqrt(azar)) / 100;
               float poppre_aux=0;
         poppre_aux = mapmin[j] + (azar * (mapmax[j] - mapmin[j]));
         poppre[contador2] = poppre_aux;// poppre_aux;
               contador2++;
              }//for
        contador++;
     }//for

 contador = 0;
 for (int i = 0; i < NP_round; i++){//for
     for (int j = 0; j < D; j++){//for
         if (poppre[contador] < mapmin[j]){//if
                         poppre[contador] = mapmin[j];
                         }//if
         if (poppre[contador] > mapmax[j]){//if
```

```
                    poppre[contador] = mapmax[j];
                    }//if
               contador++;
          }//for
          printf("\n");
      }//for

float *poblacion;
poblacion = (float*) malloc((NP_round * (D + 1)) * sizeof(float));
contador = 0;
contador2 = 0;



for (int i = 0; i < NP_round; i++){//for
    for (int j = 0; j < D + 1; j++){//for
       if (j == 0){

              poblacion[contador + contador2] = cost[contador];

              contador++;

       } else{

              poblacion[contador + contador2] = poppre[contador2];

              contador2++;
              }

       }//for

          printf("\n");
```

```
 }//for



//*******************************************************************************
*********************************************************************************
**********************************************

//---------------declaramos e inicializamos a 0 dist_3d-----------------

//-----------------------------------------------------------------------

float dist_3d [num_barridos][num_medidas];

	for (int i1=0; i1<num_barridos; i1++){

		for (int i2=0; i2<num_medidas; i2++){

			dist_3d [i1][i2]= 0;

		}

	}

float *dev_dist_3d;




//*******************************************************************************
*********************************************************************************
*****************************************

//---------------declaramos e inicializamos a 0 cart_3d--------------------------
-----

//-------------------------------------------------------------------------------
-----

float cart_3d [num_barridos*num_medidas][3];

	for (int i3=0; i3<(num_barridos*num_medidas); i3++){

		for (int i4=0; i4<3; i4++){

			cart_3d [i3][i4]= 0;

		}

	}

float *dev_cart_3d;
```

```
float *real_3d;

real_3d = (float*) malloc((num_barridos*num_medidas) * sizeof(float));


//Comprobaciónes de valores de posición dentro de los límites

if (posicion[0] < 0){ ERROR01 += 1;     }

if (posicion[0] >= m)      { ERROR01 += 1;     }

if (posicion[1] < 0){ ERROR01 += 1;     }

if (posicion[1] >= n)      { ERROR01 += 1;     }

if (posicion[2] < 0){ ERROR01 += 1;     }

if (posicion[2] >= o)      { ERROR01 += 1;     }

if (Mapa_3D[host_x_round][host_y_round][host_z_round] == 1)  {   ERROR01   +=   1;
       }




CudaSafeCall( cudaMalloc( (void**)&dev_posicion, 4 * sizeof(float)), "cudaMalloc"
);


CudaSafeCall( cudaMalloc( (void**)&dev_theta, sizeof(float)), "cudaMalloc" );


CudaSafeCall(  cudaMalloc(  (void**)&dev_mapmax,  4  *  sizeof(float)),  "cudaMalloc"
);


CudaSafeCall(  cudaMalloc(  (void**)&dev_mapmin,  4  *  sizeof(float)),  "cudaMalloc"
);


CudaSafeCall(   cudaMalloc(   (void**)&dev_dist_3d,   (num_barridos*num_medidas)   *
sizeof(float)), "cudaMalloc" );


CudaSafeCall( cudaMalloc( (void**)&dev_cart_3d, (num_barridos*num_medidas) * 3 *
sizeof(float)), "cudaMalloc" );


CudaSafeCall( cudaMalloc( (void**)&dev_datos, 2400 * sizeof(int) ), "cudaMalloc"
);
```

```
if (ERROR01!=0) {

printf("ERROR: Valores de posicion fuera de rango\n");

}

else{

//--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

//A PARTIR DE AQUI PROGRAMA!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!------------------------------------------------------------------------------------------------------------------------------------------------------------

//---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

//cudaMemcpy

CudaSafeCall( cudaMemcpy( dev_posicion, posicion, 4 * sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_theta, &theta, sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_mapmax, mapmax, 4 * sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_mapmin, mapmin, 4 * sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_dist_3d, dist_3d, (num_barridos*num_medidas) * sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_cart_3d, cart_3d, (num_barridos*num_medidas) * 3 * sizeof(float), cudaMemcpyHostToDevice ));
```

```
CudaSafeCall( cudaMemcpy( dev_datos, datos, 2400 * sizeof(int),
cudaMemcpyHostToDevice ));


//FIN MEMCOPY--------------------------------------------------------------------
-------------------




//llamada kernel-----------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------


//kernel<<<blocks, threads>>>


kernel<<<num_barridos,num_medidas>>>( dev_theta, dev_mapmin, dev_mapmax,
dev_posicion, dev_cart_3d, dev_dist_3d, dev_datos );


//SE COMPRUEBA QUE SE EJECUTA EL KERNEL CORRECTAMENTE

CudaCheckError();//("Kernel Execution Failed!");




CudaSafeCall( cudaMemcpy( dist_3d, dev_dist_3d, (num_barridos * num_medidas) *
sizeof(float), cudaMemcpyDeviceToHost ));

CudaSafeCall( cudaMemcpy( cart_3d, dev_cart_3d, (num_barridos * num_medidas) * 3
* sizeof(float), cudaMemcpyDeviceToHost ));




//*******************************************************************************
*************************************

//***COPIA MATRIZ KERNEL AL MATRIZ REAL

//*******************************************************************************
*************************************




int contaFitness = 0;

for (int i1=0; i1<num_barridos; i1++){
```

```
        for (int i2=0; i2<num_medidas; i2++){

                real_3d[contaFitness] = dist_3d [i1][i2];

                contaFitness ++;

        }

}


//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//prueba dist 3d

float dist_3db [num_barridos][num_medidas];

        for (int i1=0; i1<num_barridos; i1++){

                for (int i2=0; i2<num_medidas; i2++){

                        dist_3db [i1][i2]= 0;

                }

        }


FILE* mD3DA;

if((mD3DA=fopen("matrizDist3DA.txt", "w"))==NULL){

puts("Cannot open file");

}

for(int ii=0; ii<num_barridos; ii++){

        for(int jj=0; jj<num_medidas; jj++){

                fprintf(mD3DA, "%5.5f", dist_3db[ii][jj]);

        }
```

```
        fprintf(mD3DA, "\n" );

}

fclose(mD3DA);

for (int i1=0; i1<num_barridos; i1++){

        for (int i2=0; i2<num_medidas; i2++){

                dist_3db [i1][i2]= dist_3d [i1][i2];

        }

}


FILE* mD3DB;

if((mD3DB=fopen("matrizDist3DB.txt", "w"))==NULL){

puts("Cannot open file");

}

for(int ii=0; ii<num_barridos; ii++){

        for(int jj=0; jj<num_medidas; jj++){

                fprintf(mD3DB, "%5.5f", dist_3db[ii][jj]);

        }

        fprintf(mD3DB, "\n" );

}

fclose(mD3DB);




//--------------------------------------------------------------------------------
-------------------------

//ESCRIBO DISTD3D.

//--------------------------------------------------------------------------------
-------------------------

FILE* mD3D;

if((mD3D=fopen("matrizDist3D.txt", "w"))==NULL){

puts("Cannot open file");

}

for(int ii=0; ii<num_barridos; ii++){
```

```
        for(int jj=0; jj<num_medidas; jj++){

                fprintf(mD3D, "%5.5f ", dist_3d[ii][jj]);

        }

        fprintf(mD3D, "\n" );

}

fclose(mD3D);


//--------------------------------------------------------------------------------
------------------------

//ESCRIBO CART3D

//--------------------------------------------------------------------------------
------------------------

FILE* mCart3D;

if((mCart3D=fopen("matrizCart3D.txt", "w"))==NULL){

puts("Cannot open file");

}

for(int iii=0; iii<num_barridos*num_medidas; iii++){

        for(int jjj=0; jjj<3; jjj++){

                fprintf(mCart3D, " %5.5f", cart_3d[iii][jjj]);

        }

        fprintf(mCart3D, "\n" );

}

fclose(mCart3D);



//--------------------------------------------------------------------------------
-----------------------------------------------------------

//FIN CREACION MATRIZ REAL
```

```
//--------------------------------------------------------------------------------
----------------------------------------------------------------

//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//********************************************************************************
**********************************************************************************
****

//

//

//AQUI EMPIEZA EL BUCLE ANTES DEL WHILE

//


       for(int i = 0; i < NP_round; i++){


        //variables------------------


       posicion[0] = poblacion[((i*5) + 1)];

       posicion[1] = poblacion[((i*5) + 2)];

       posicion[2] = poblacion[((i*5) + 3)];

       posicion[3] = poblacion[((i*5) + 4)];

       host_x_round = (int) posicion[0];

       host_y_round = (int) posicion[1];

       host_z_round = (int) posicion[2];

       theta = (posicion[3] + 180);


       //fin redifinicion---
```

```
//cudaMemcpy

CudaSafeCall( cudaMemcpy( dev_posicion, posicion, 4 * sizeof(float), cudaMemcpyHostToDevice ));

CudaSafeCall( cudaMemcpy( dev_theta, &theta, sizeof(float), cudaMemcpyHostToDevice ));

//No es necesario volver a hacer el resto de cudaMemcpy

//FIN MEMCOPY---------------------------------------------------------------------------------

//llamada kernel-----------------------------------------------------------------------------------------------------------------------------------------------------------------

	//kernel<<<blocks, threads>>>

			kernel<<<num_barridos,num_medidas>>>( dev_theta, dev_mapmin, dev_mapmax, dev_posicion, dev_cart_3d, dev_dist_3d, dev_datos );

			CudaCheckError();//("Kernel Execution Failed!");

	CudaSafeCall( cudaMemcpy( dist_3d, dev_dist_3d, (num_barridos * num_medidas) * sizeof(float), cudaMemcpyDeviceToHost ));

	CudaSafeCall( cudaMemcpy( cart_3d, dev_cart_3d, (num_barridos * num_medidas) * 3 * sizeof(float), cudaMemcpyDeviceToHost ));
```

```
	//fin kernel 2

//******************************************************************************
************************************

//***COPIA MATRIZ KERNEL AL MATRIZ REAL

//******************************************************************************
************************************


float *estimada;

estimada = (float*) malloc((num_barridos*num_medidas) * sizeof(float));

int contaFitness = 0;

for (int i1=0; i1<num_barridos; i1++){

	for (int i2=0; i2<num_medidas; i2++){

		estimada[contaFitness] = dist_3d [i1][i2];

		contaFitness ++;

	}

}


//******************************************************************************
************************************


	for (int i1=0; i1<num_barridos; i1++){

		for (int i2=0; i2<num_medidas; i2++){

			dist_3db [i1][i2]= 0;

		}

	}


	FILE* mD3DAsegunda;


	if((mD3DAsegunda=fopen("matrizDist3DAsegunda.txt", "w"))==NULL){
```

```
        puts("Cannot open file");
}
for(int ii=0; ii<num_barridos; ii++){
        for(int jj=0; jj<num_medidas; jj++){

                fprintf(mD3DAsegunda, "%5.5f ", dist_3db[ii][jj]);

        }
        fprintf(mD3DAsegunda, "\n" );
}
fclose(mD3DAsegunda);

for (int i1=0; i1<num_barridos; i1++){
        for (int i2=0; i2<num_medidas; i2++){
                dist_3db [i1][i2]= dist_3d [i1][i2];
        }
}

FILE* mD3DBsegunda;

if((mD3DBsegunda=fopen("matrizDist3DBsegunda.txt", "w"))==NULL){
        puts("Cannot open file");
}
for(int ii=0; ii<num_barridos; ii++){
        for(int jj=0; jj<num_medidas; jj++){

                fprintf(mD3DBsegunda, "%5.5f ", dist_3db[ii][jj]);

        }
        fprintf(mD3DBsegunda, "\n" );
}
```

```
	fclose(mD3DBsegunda);



	//----------------------------------------------------------------------------
--------------------------------
	//ESCRIBO DISTD3D.
	//----------------------------------------------------------------------------
--------------------------------
	FILE* mD3Dsegunda;

	if((mD3Dsegunda=fopen("matrizDist3Dsegunda.txt", "w"))==NULL){
		puts("Cannot open file");
	}
	for(int ii=0; ii<num_barridos; ii++){
		for(int jj=0; jj<num_medidas; jj++){

			fprintf(mD3Dsegunda, "%5.5f ", dist_3d[ii][jj]);

		}
		fprintf(mD3Dsegunda, "\n" );
	}
	fclose(mD3Dsegunda);

	//----------------------------------------------------------------------------
-------------------------------
	//ESCRIBO CART3D
	//----------------------------------------------------------------------------
-------------------------------
	FILE* mCart3Dsegunda;

	if((mCart3Dsegunda=fopen("matrizCart3Dsegunda.txt", "w"))==NULL){
		puts("Cannot open file");
```

```
        }
        for(int iii=0; iii<num_barridos*num_medidas; iii++){
               for(int jjj=0; jjj<3; jjj++){

                      fprintf(mCart3Dsegunda, " %5.5f", cart_3d[iii][jjj]);

               }
               fprintf(mCart3Dsegunda, "\n" );
        }
        fclose(mCart3Dsegunda);




        poblacion[(i*5)] = fitness_3d(estimada, real_3d);



        free(estimada);
        }//FOR CON POBLACION



        //hasta aqui


//
//*******************************************************************************
*********************************************************************************
****
//*******************************************************************************
*********************************************************************************
****
//*******************************************************************************
*********************************************************************************
****
```

```
//*******************************************************************************
********************************************************************************
****

FILE* mPoblacion;

	if((mPoblacion=fopen("poblacion.txt", "w"))==NULL){
		puts("Cannot open file");
	}
	for(int iii=0; iii<5; iii++){
		for(int jjj=0; jjj<5; jjj++){

			fprintf(mPoblacion, " %5.5f ", poblacion[((iii*5)+jjj)]);

		}
		fprintf(mPoblacion, "\n" );
	}
	fclose(mPoblacion);

}//if (ERROR01!=0)------> selecciona mostrar error o ejecutar
//TERMINA EJECUCIÓN

//*******************************************************************************
********************************************************************************
****
//*******************************************************************************
********************************************************************************
****
//WHILE
//*******************************************************************************
********************************************************************************
****
```

```
//*******************************************************************************
*********************************************************************************
****


//Variables dentro del While

int count = 1;

int iter_max = 40; //Por defecto

int count_a = 0;

int count_b = 0;

int count_c = 0;

float error = 10000;

float error_max = 100000;

float dif_errores = 100000;


float F = 0.85;

float trial[1][D]; //Trial de (1,D)

for (int i=0; i<D; i++){

       trial[0][i] = 0;

       }

float bestmem[1][5];//D+1 = 5

for (int i=0; i<D +1; i++){

       bestmem[0][i] = 0;

       }

float pob_aux[25][5];

for (int i=0; i<25; i++){

       for(int j=0; j<5; j++){

       pob_aux[i][j] = 0;

              }

       }

float error_med = 0;

float error_aux = 0;

float error_max_aux = 0;
```

```
float dif_errores_aux = 0;
float error_global = 0;
float error_trial = 0;
int a = 0;
int b = 0;
int c = 0;

while ((count <= iter_max) && (error != error_max) && (error_max > 3*NP/2)){
            //while hasta el final del algoritmo (con count +1)
            for (int i = 0; i<NP ; i++){
                  //Se eligen aleatoriamente 3 vectores distintos de i
                  a = (rand() % 25);
                  while ((a==i)||(a==0)){
                        a = (rand() % 25);
                        }//while

                  b = (rand() % 25);
                  while ((b==i)||(b==a)||(b==0)){
                        b = (rand() % 25);
                        }//while

                  c = (rand() % 25);
                  while ((c==i)||(c==a)||(c==b)||(c==0)){
                        c = (rand() % 25);
                        }//while

                  //MUTACION Y CRUCE

                  for (int k=1; k<(D+1); k++){
                        int cross_rand = (rand() % 100);
```

```
                        if (cross_rand < (100*CR)){
                                trial[0][(k-1)]   =   poblacion[((c*5)+k)]   +
F*(poblacion[((a*5)+k)] - poblacion[((b*5)+k)]);//poblaciones en lista
                                }
                        else {
                        trial[0][(k-1)]=poblacion[((i*5)+k)];
                        }

                        //Si nos salimos del mapa nos quedamos en el borde
                        if(trial[0][(k-1)] < mapmin[(k-1)]){
                                trial[0][(k-1)] = mapmin[(k-1)];
                        }
                        if(trial[0][(k-1)] > mapmax[(k-1)]){
                                trial[0][(k-1)] = mapmax[(k-1)];
                        }
                }//for k<(D+1)

                //SELECCION

                posicion[0] = trial[0][0]; //poblacion[((i*5) + 1)];
                posicion[1] = trial[0][1]; //poblacion[((i*5) + 2)];
                posicion[2] = trial[0][2]; //poblacion[((i*5) + 3)];
                posicion[3] = trial[0][3]; //poblacion[((i*5) + 4)];
                theta = (posicion[3] + 180);

                CudaSafeCall(  cudaMemcpy(  dev_posicion,  posicion,  4  *
sizeof(float), cudaMemcpyHostToDevice ));
                CudaSafeCall( cudaMemcpy( dev_theta, &theta, sizeof(float),
cudaMemcpyHostToDevice ));

                                kernel<<<num_barridos,num_medidas>>>(
dev_theta,  dev_mapmin,  dev_mapmax,  dev_posicion,  dev_cart_3d,  dev_dist_3d,
dev_datos );
```

```
                    CudaCheckError();//("Kernel Execution Failed!");


                    CudaSafeCall( cudaMemcpy( dist_3d, dev_dist_3d, (num_barridos
* num_medidas) * sizeof(float), cudaMemcpyDeviceToHost ));

                    CudaSafeCall( cudaMemcpy( cart_3d, dev_cart_3d, (num_barridos
* num_medidas) * 3 * sizeof(float), cudaMemcpyDeviceToHost ));


                    float *estimada;

                    estimada   =   (float*)   malloc((num_barridos*num_medidas)   *
sizeof(float));

                    int contaFitness = 0;

                    for (int i1=0; i1<num_barridos; i1++){

                           for (int i2=0; i2<num_medidas; i2++){

                                  estimada[contaFitness] = dist_3d [i1][i2];

                                  contaFitness ++;

                                  }

                           }

                    //Fitness_3d devuelve el error de las medidas para el pto.
perturbado


                    error_trial = fitness_3d(estimada, real_3d);

                    //Si   el   error   del   elemento   mutado   es   menor,   serà   el
integrante de la nueva generacion



                           if(error_trial < poblacion[(i*5)]){

                                  for (int j=1; j <(D+1); j++){

                                         pob_aux[i][j]  =  trial[0][j-1];//Meto  la
Nueva gen en una auxiliar para no mezclar gen

                                         }

                                  pob_aux[i][0] = error_trial;

                           }//if(error_trial...
```

```
                    else{
                           for (int j=0; j <(D+1); j++){
                                  pob_aux[i][j]                          =
poblacion[((i*5)+j)];//meto la nueva generacion en una auxiliar para no mezclar
gen
                           }
                    }//else if error_trial....
             //if verion==1
             free(estimada);
      }//end for i=1:NP
      //la poblacion pasa a la nueva generacion


      int copia_poblacion = 0;
      for(int i_copy = 0; i_copy < 25; i_copy++){
             for(int j_copy = 0; j_copy < 5; j_copy++){
                    poblacion[copia_poblacion]=pob_aux[i_copy][j_copy];
                    copia_poblacion++;
             }
      }
      //DISCARDING


      //Se busca el mejor elemento, el peor error y el global en cada
generacion
      error_aux=poblacion[0];
      for (int j=0; j < (D+1); j++){
             bestmem[0][j]=poblacion[j];
      }
      error_med=(error_aux/(num_medidas*num_barridos));//error  medio  por
medida en celdillas
      if(abs(error_aux - error) < 1){//contador que determina el crit de
conv
             count_b= count_b +1;
      }
```

```
        else{
                count_b= 0;
        }//if(abs...)
        error = error_aux;


        //Calcula MAX de error_max_aux
        for(int i=0; i < NP_round; i++){
                if(error_max_aux < poblacion[(i*5)]){
                        error_max_aux = poblacion[(i*5)];
                }
        }


        if (abs(error_max - error_max_aux) < 1){//contador que determina un
criterio de convergencia
                count_a = count_a + 1;
        }
        else{
                count_a = 0;
        }//if
        error_max = error_max_aux;
        dif_errores_aux = error_max - error;
        if(abs(dif_errores_aux - dif_errores) < 1){//contador
                count_c = count_c + 1;
        }
        else{
                count_c = 0;
        }//if
        dif_errores = dif_errores_aux;
        for(int i = 0; i < NP_round; i++){
                 error_global = error_global + poblacion[(i*5)];
        }//error_global = sum(poblacion(:,1);
```

```
		//CONDICION DE CONVERGENCIA, CUANDO NOS ACERCAMOS LIMITAMOS EL AREA
		if (error - error_max < 1){
			F=0.2;
		}

		//CONDICION DE CONVERGENCIA, SI SE HA LLEGADO DISMINUIMOS LA POBLACION
		if (error - error_max == 0){
		NP=10;
		}//if

		count = count + 1;

		}//WHILE

printf("\n");
printf("Count vale: %d\n", count);
printf("Error vale: %f\n", error);
printf("Error_max vale: %f\n", error_max);
printf("Error_global vale: %f\n", error_global);
printf("Error_med vale: %f\n", error_med);
printf("poblacion 0,1 vale: %f\n", poblacion[1]);
printf("poblacion 0,2 vale: %f\n", poblacion[2]);
printf("poblacion 0,3 vale: %f\n", poblacion[3]);
printf("poblacion 0,4 vale: %f\n", poblacion[4]);

//****************************************************
//****************************************************
//****************************************************
```

```
//-----------------------------------------------------------------------------
--------------------

//FIN DEL PROGRAMA

//MODIFICADO DEL ORIGINAL PARA HACER EL GENETICO

//

//-----------------------------------------------------------------------------
--------------------



              //LIBERAR  LA  MEMORIA--------------------------------------------------
-------------------------------------------------------------------------------
--

       //


       CudaSafeCall( cudaFree( dev_theta ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_mapmin ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_mapmax ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_posicion ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_cart_3d ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_dist_3d ), "cudaFree" );


       CudaSafeCall( cudaFree( dev_datos ), "cudaFree" );


cudaDeviceReset ();//Explicitly destroys and cleans up all resources associated
with the current device in the current process. Any subsequent API call to this
device will reinitialize the device.

free(poppre);
```

```
free(cost);

free(poblacion);

free(real_3d);


printf("Tiempo transcurrido: %f", ((double)clock() - start) / CLOCKS_PER_SEC);

getchar();// PULSAR ENTER PARA TERMINAR

return 0;


} //MAIN
```

Bibliografía

[1] Bagley, J. D. (1967). "The Behavior of adaptative systems wich employ genetic and correlation algorithms." (Doctoral diseertation, University of Michigan). Disseration Abstracts Internationa, 29(12), 5106B. (University Microfilms No. 68-7556).

[2]Burgard, W., Fox, D., Henning, D., Schmidt, T. (Aug 4–8, 1996). "Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids," Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-96), Portland, Oregon .

[3] Cavicchio, D. J. (1970). "Adaptative search using simulated evolution." Unpublished doctoral dissertation, University of Michigan, Ann Arbor.

[4] Ellis, Margaret A., Stroustrup, Bjarne. (1990). "The Annotated C++ Reference Manual." Addison-Wesley.

[5] Farber, Rob. (2011). "CUDA Application Design and Development." Morgan Kaufmann.

[6] Fox, D., Burgard, W., Thrun, S. (1999). "Markov localization for mobile robots in dynamic environments". J. Artif. Intell. Res. 11(11), 391-427.

[7] Goldberg, David E. (1989). "Genetic Algorithms in Search, Optimization, and Machine Learning." Addison-Wesley.

[8] Ho, N., Jarvis, R. (Sep. 2008). "Vision-Based Global Localisation Using a 3D Environmental Model Created by a Laser Range Scanner," Proceedings of the International Conference on Intelligent Robots and Systems (IROS'08), Acropolis Convention Center, Nice, France.

[9] Kümmerle, R., Triebel, R., Pfaff, P., Burgard,W. (2008). "Monte Carlo localization in outdoor terrains using multi-level surface maps," J. Field Robot. **42**, 213–222.

[10] Kirk, David B., Hwu, Wen-mei W. (2010). "Programming Massively Parallel Processors. A Hands-on Approach." Morgan Kaufmann.[10] Kümmerle, R., Triebel, R., Pfaff, P., Burgard,W. (2008). "Monte Carlo localization in outdoor terrains using multi-level surface maps," J. Field Robot. **42**, 213–222.

[11] Lingemann, K., Nüchter, A., Hertzberg, J., Surmann, H. (2005). "High-speed laser localization for mobile robots". Robot. Auton. Syst. **51**, 275–296.

[12] Martín, Fernando., Moreno, Luis., Garrido, Santiago., Blanco, Dolores. (2011). "High-accuracy global localization filter for three-dimensional environments." Cambridge University Press.

[13] Martin, Robert C. (2009). "Código Limpio. Manual de estilo para el desarrollo ágil de software." Prentice Hall.

[14] Nüchter, A., Lingemann, K., Hertzberg, J. (2007). "6D slam – 3D mapping outdoor environments," J. Field Robot. **24**, 699–722.

[15] NVIDIA®. (2010). "Compute Visual Profiler User Guide." http://www.nvidia.es

[16] NVIDIA® CUDA™. (2011). "NVIDIA CUDA C Programming Guide (versión 4.0)." http://www.nvidia.es

[17] NVIDIA®. (2010). "NVIDIA GF100. World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism". http://www.nvidia.es

[18] NVIDIA®. (2012). "NVIDIA GeForce GTX 680 Whitepaper. The fastest, most efficient GPU ever built." http://www.nvidia.es

[19] Parker, J.R. (2011). "Algorithms for Image Processing and Computer Vision." Wiley Publishing, Inc.

[20] Rosenberg, R. S. (1967). "Simulation of genetic populations with biochemical properties." (Doctoral dissertation, University of Michigan). Dissertation Abstracts International, 28(7), 2732B. (University Microfilms No. 67-17836).

[21] Royer, E., Lhuillier, M., Dhome, M., Lavest, J.-M. (2007). "Monocular vision for mobile robot localization and autonomous navigation," Int. J. Comput. Vis. **74**, 237–260.

[22] Sanders, Jason., Kandrot, Edward. (2011). "CUDA By Example. An Introduction to General-Purpose GPU Programming." Addison-Wesley.

[23] Se,S., Lowe, D. G., Little, J. J. (2005). "Vision-based global localization and mapping for mobile robots," IEEE Trans. Robot. **21**, 3.

[24] Thrun, S., Hähnel, D., Ferguson, D., Montemerlo, M., Triebel, R., Burgard, W., Baker, C., Omohundro, Z., Thayer, S., Whittaker, W. (May 12–17, 2003). "A System for Volumetric Robotic Mapping of Abandoned Mines," Proceedings of the International Conference on Robotics and Automation (ICRA'03).

[25] Triebel, R., Pfaff, P., Burgard, W. (Oct. 2006). "Multi-Level Surface Maps for Outdoor Terrain Mapping and Loop Closing," Proceedings of the International Conference on Intelligent Robots and Systems (IROS'06), Beijing, China.

[26] Tsai H.-H., Lai, C.-C. Lin, S.-W. (2005). "Multisensor 3D posture determination of a mobile robot using inertial and ultrasonic sensors," J. Intell. Robot. Syst. **42**, 317–335.

[27] Vahdat,A. R., Ashrafoddin, N. N., Ghidary, S. S. (Sep. 25–28, 2007). "Mobile Robot Global Localization using Differential Evolution and Particle Swarm Optimization," Proceedings of the Congress on Evolutionary Computation (CEC'07).

[28] Weinberg, R. (1970). "Computer simulation of a living cell." (Doctoral dissertation, University of Michigan). Dissertations Abstracts International, 31(9), 5312B. (University Microfilms No. 71-4766).