Resolución de Laberinto con procesamiento de imágenes virtuales

Ezequiel Focaraccio, Luciano Galluzzo, Agustín Riva

[1]Universidad Nacional de La Matanza,
Departamento de Ingeniería e Investigaciones Tecnológicas,
Florencio Varela 1903 - San Justo, Argentina
eafocaraccio@gmail.com, luchogalluzzo421@gmail.com, aagusriva@gmail.com

Resumen. El presente trabajo de investigación tiene como objetivo poder comprobar la existencia de una posible resolución de un laberinto basado líneas planas previo a su recorrido. El método consiste en tomar imágenes de este y analizar, con técnicas de HPC, las distintas rutas existentes desde cualquier inicio hasta el fin y obtener los caminos más cortos.

Palabras claves: GPU, HPC, MPI, OpenMP, Seguidor de Línea, Robot, Laberinto, Análisis Fotográfico, Detección de colores

1 Introducción

LaberintoSmart es un proyecto basado en un seguidor de líneas que ofrece la posibilidad de resolver un laberinto a medida que el robot va recorriendo el camino. Además, luego de haber encontrado la salida, permite ejecutar una versión optimizada del camino resuelto, es decir, evitando loops y tiempos muertos.

En este documento, profundizaremos la posibilidad de poder obtener la resolución del laberinto a través de un procesamiento de imágenes virtuales en tiempo real, para luego calcular, previo a la ejecución del robot, el camino más optimizado hasta el final del laberinto desde cada una de las entradas de este hasta el final.

Actualmente contamos con librerías como OpenCV (Computer Vision) para el procesamiento de imágenes y detección de objetos en ella, y con librerías como TensorFlow utilizada para aplicaciones que requieran técnicas de aprendizaje automático a través de redes neuronales. Además, existen aplicaciones que detectan distintos colores en imágenes como ColorID desarrollada por Gordon L. Hempton. En nuestro caso usaremos los métodos brindados por la librería OpenCV para detectar objetos, buscando detectar las líneas (de un color que contrasta claramente con el resto de la imagen) que conforman nuestro laberinto.

2 Desarrollo

Lo primero que necesitamos para llevar a cabo el desarrollo de nuestra funcionalidad es una cámara con definición 4K UHD colocada en el robot, que tenga la altura suficiente para tomar una gran cantidad de fotografías del laberinto para luego unirlas y formar una sola imagen panorámica. Para eso colocaremos al auto en el centro del laberinto y la cámara tomará alrededor de 360 fotos, desde el ángulo 0° a 359° inclusive.

Por defecto, las imágenes en resolución 4K UHD son en realidad 4 imágenes HD (1920x1080) que se unen y se procesan en paralelo para obtener una resolución lineal al doble que el estándar de 1080p. Además, generan que la superficie de la imagen se cuadruplique.

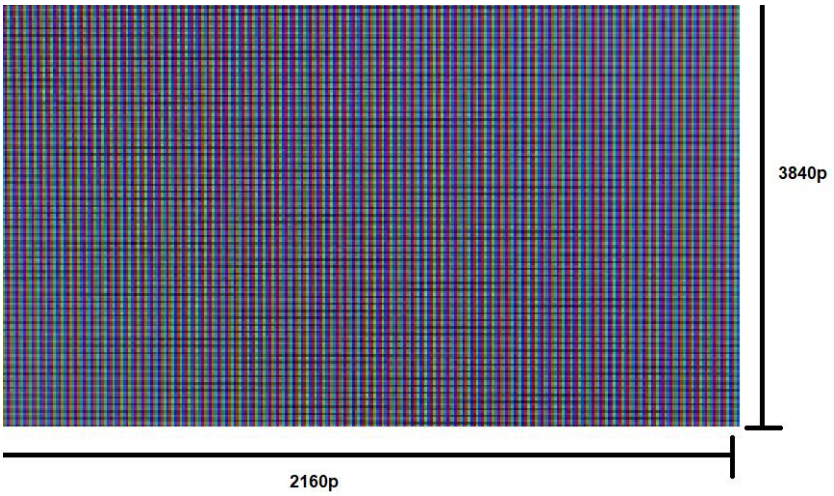
4K UHDV (3840x2160)	
1080p (1920x1080)	1080p (1920x1080)
1080p (1920x1080)	1080p (1920x1080)

Una vez obtenidas las imágenes y almacenadas en la memoria virtual, se procederá a dividir cada una en 3840 “tiras” de píxeles de 2160 pixeles cada tira.

Cada una de estas tiras de píxeles serán tomadas por un proceso diferente, en un entorno de memoria distribuida en donde todos los procesos de tiras se ejecutan en paralelo junto con la biblioteca MPI de paso de mensajes.

Luego de obtenida esta división, cada proceso se subdividirá en 2160 procesos en donde cada uno ejecutará 1 pixel diferente. Esta segunda división se realizará mediante un entorno de memoria compartida ya que todos estos píxeles forman la tira de píxeles dividida anteriormente. Claro está, que dicha subdivisión se realizará junto con directivas del compilador y utilizando la interfaz de multiprocesos OpenMP.

Una vez logrado esto, se habrá generado una especie de “mapa virtual” en donde cada pixel está dividido en un proceso, para poder formar así el procesamiento de la imagen.



Una vez obtenido este mapa virtual, se procederá a analizar el contenido de cada una con las funciones que nos brinda OpenCV y TensorFlow.

OpenCV es una librería que puede ser utilizada para múltiples fines debido a que se encuentra bajo una licencia BSD, lo que permite el uso y personalización de su código para cada caso en particular. En términos generales, se suele utilizar para la detección de objetos, detección de movimiento e incluso reconocimiento de patrones.

En cuanto a TensorFlow, es una librería de código abierto desarrollada por Google para construir y entrenar redes neuronales. El objetivo es desarrollar un aprendizaje automático del sistema permitiendo resolver patrones y correlaciones, relacionadas con su propósito, cada vez más complejos.

En cada imagen detectaremos los “objetos” que encuentra OpenCV (En nuestro caso las líneas) y calculando distancias, tamaños y uniones de estas (que representan los cruces del laberinto) construiremos en memoria un laberinto.

Con el mismo algoritmo utilizado en nuestro trabajo para la resolución de laberintos procederemos a encontrar la solución para llegar al punto final del mismo desde cada uno de los puntos iniciales. El punto final debe ser previamente determinado para que el procesamiento del reconocimiento de imagen pueda detectarlo.

Por último, una vez resuelto el laberinto, se realizará una pasada para cada camino obtenido con el algoritmo de optimización y así obtener finalmente los caminos más cortos. Luego de esto, el programa enviará al Arduino las indicaciones (Seguir adelante, doblar a izquierda/derecha, entre otras) para ejecutar el laberinto de la mejor manera posible en el primer intento.

3 Explicación del algoritmo.

```
angulo = ['0','1','2',...,'359'];
for(i=0;i<angulo.length();i++){
    motores.moverCamara(angulo[i]);
    imagen[i] = camara.SacarFoto(angulo[i]);
    mapa_virtual.add(paralelizarImagen(imagen[i],RESOLUCION_4KUHD));
}

if(mapa_virtual.state() == SUCESS){
    while(obtenerLaberinto()){
        resolver_laberinto();
    }
    optimizar_laberinto();
}
```

Cabe aclarar ante todo que este es solo una parte del algoritmo completo para la resolución del problema presentado en este documento. Dicho esto, vamos a identificar cada método y variable del código para que sea entendible de la mejor manera:

- *ángulo*: Representa a un array donde posee todos los ángulos en los que deberá moverse el robot para capturar una imagen.
- *moverCamara*: En este método se invoca a funciones como *digitalWrite* y *analogWrite* para poner en funcionamiento a los motores del robot y que se genere el giro deseado para la posición requerida.
- *SacarFoto*: En ese método se enciende la cámara del robot y ejecuta la funcionalidad de capturar una imagen.
- *mapa_virtual*: En él se irán agregando los diferentes bits que devuelve cada proceso luego de serializar cada tira de pixel.
- *paralelizarImagen*: En este método se ejecutan las funciones para crear la paralelización con memoria distribuida en cada tira de píxel mediante MPI y las directivas del compilador para paralelizar con memoria compartida a cada píxel mediante OpenMP. Aquí es donde entra en juego la GPU.
- *obtenerLaberinto*: En este método se ejecutan las funciones de procesamiento de imágenes que proporcionan las librerías OpenCV y el motor de procesamiento TensorFlow.
- *resolver_laberinto*: En este método se obtiene el camino exitoso de cada entrada hasta el fin del laberinto.
- *optimizar_laberinto*: En este método se obtiene el camino más corto para llegar a la salida del laberinto.

4 Pruebas que pueden realizarse

Las pruebas que podemos realizar para probar esta funcionalidad es colocar el auto en el medio del laberinto para que el auto tome las 360 fotografías y que luego, colocando el auto en el inicio del laberinto, logre resolverlo eligiendo el camino más óptimo en el primer intento.

Otra prueba fundamental para validar el funcionamiento es colocar al auto en un laberinto que sea imposible de resolver. De esa manera debemos comprobar que el programa no detecte una solución del laberinto y pueda detectar la imposibilidad de resolverlo.

Esto permitirá obtener datos como: el tiempo que tarda el programa en resolver el procedimiento y el resultado que arroje.

5 Conclusiones

En resumen, a través de esta investigación se logró demostrar la veracidad de lo supuesto: existe la forma en la que, mediante implementación del concepto de HPC y las metodologías ofrecidas por GPU, se pueda evitar la pérdida de tiempo en ver si es posible lograr resolver el laberinto y, aún más, optimizarlo.

Durante la documentación logramos profundizar nuestros conocimientos del procesamiento de imágenes y conocer la existencia de librerías y motores ya predefinidos que facilitan aún más la tarea de la programación.

Es muy probable que en un futuro los componentes de imagen y video evolucionen de una manera abismal por lo que sería necesario investigar métodos y conceptos para poder involucrar aún más nuestro robot al mundo de las redes neuronales.

6 Referencias

1. Brunner, R.; Doepke, F.; Laden, B.: Chapter 26. Object Detection by Color: Using the GPU for Real-Time Video Image Processing. GPU Gems 3 (2007). https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch26.html
2. Representar y resolver un laberinto con una imagen. (2017). http://programandonet.com/questions/812/representar-y-resolver-un-laberinto-con-una-imagen
3. Kurtaev, Dm: TensorFlow Object Detection API. (2018). https://github.com/opencv/opencv/wiki/TensorFlow-Object-Detection-API
4. What is the difference between OpenCV and Tensorflow? (2017). https://www.quora.com/What-is-the-difference-between-OpenCV-and-Tensorflow
5. OpenCV. (2019). https://opencv.org/cvpr-2019-tutorial-html/
6. Pakdaman, M.; Sanaatiyan, M.: Design and Implementation of Line Follower Robot. (2009). https://ieeexplore.ieee.org/abstract/document/5380235/authors#authors
7. Resolución 4K. (2019). https://es.wikipedia.org/wiki/Resoluci%C3%B3n_4K