Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estudio, optimización y comparativa de algoritmos de procesamiento de nubes de puntos para detección de obstáculos desde UAVs

Autor: Pablo Jiménez Cámara

Tutor: Aníbal Ollero Baturone

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estudio, optimización y comparativa de algoritmos de procesamiento de nubes de puntos para detección de obstáculos desde UAVs

Autor:
Pablo Jiménez Cámara

Tutor:
Aníbal Ollero Baturone
Catedrático

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado: Estudio, optimización y comparativa de algoritmos de procesamiento de nubes de puntos para detección de obstáculos desde UAVs

Autor: Pablo Jiménez Cámara
Tutor: Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Querría empezar dándole las gracias a mi familia y mis amigos por darme una vida llena de buenos momentos, y en los no tan buenos agradecerles por haber intentado siempre subirme los ánimos y sacarme una sonrisa en esta época tan dura a nivel académico y personal.

Quiero darles también las gracias a FADA-CATEC, y en concreto a Fran y Rafa, por darme la oportunidad de formarme profesionalmente de vuestra mano y crecer como persona, y a mis compañeros de unidad por crear un gran ambiente de compañerismo que nada más llegar me hizo sentirme muy cómodo. En especial quería también agradecerle a Fidel por ser el mejor tutor que un estudiante en prácticas puede tener además de ser una maravillosa persona, por estar siempre ahí cuando necesitaba de su ayuda, por tener una paciencia enorme para ayudarme con todos los errores que surgieron los primeros meses y por hacerme sentir valorado por mi esfuerzo y trabajo devolviéndome así la confianza en mí mismo que había perdido. En definitiva, gracias por todos estos meses compañeros.

Por último, quiero dedicarle este trabajo a mis abuelos Andrés y José María y a mi gran amigo Enrique, que de una forma u otra han formado parte de mi vida y me han ayudado a convertirme en la persona que soy ahora. Me hubiese gustado contaros lo feliz que me encuentro de cerrar esta etapa de mi vida, pero por cosas de la vida no me queda otra que desear haberos hecho orgullosos de lo que he conseguido hasta el momento, os llevaré siempre conmigo.

Resumen

Este trabajo va a tratar sobre el estudio de varias librerías de procesamiento de *nubes de puntos* y el desarrollo de algoritmos mediante *GPU* para el pre-procesamiento y la detección de obstáculos enfocado a *UAVs*. Para determinar los escenarios más interesantes en los que obtener métricas para el posterior análisis, se usarán implementaciones previas en sets de datos reales capturados en entornos urbanos con un *LIDAR 3D* embarcado en un UAV. Se analizará la comparativa de los resultados obtenidos tras ejecutar los algoritmos tanto en *CPU* como en *GPU*, teniendo en cuenta los tiempos de ejecución, el rendimiento y el tamaño de las nubes de puntos. Dicha comparativa se realizará no solo en un portátil, sino también en un kit de desarrollador de la familia *NVIDIA Jetson*, concretamente una *Jetson TX2*, la cual cuenta con varios procesadores CPU y una GPU y puede ser integrada en un UAV. La comparativa de los resultados en estos dos dispositivos nos permitirá analizar mejor las ventajas que supone el procesamiento con GPU y determinar los escenarios más eficientes para su uso en UAVs, teniendo en cuenta las limitaciones de los *sistemas embebidos*.

Abstract

This project is about studying some different point clouds libraries and the development of *GPU-based* algorithms for pre-processing and detecting obstacles focused on *UAVs*. In order to decide the most interesting scenes to obtain metrics for the later analysis, some previous implementations on real data sets captured at urban environments with a *3D-LIDAR* embedded into an UAV will be used. A comparison of the obtained results will be analysed, after executing the algorithms on both *CPU* and *GPU*, with regard to the execution times, the efficiency and the size of the point clouds. The aforementioned comparison will be made not only on the laptop, but also on a *NVIDIA Jetson TX2* developer kit, which has some CPU processors and a GPU and it can be embedded into an UAV. The comparison of the results between these two devices will allow us to do a better analysis of the GPU-based processing advantages and determine the most efficient scenes for its use on UAVs, taking into account the *embedded systems* limitations.

Índice Abreviado

Índice

Índice de Figuras

Índice de Tablas

Índice de Códigos

1 Introducción

En la vida cotidiana, los seres humanos gozamos de una serie de sentidos que nos permiten recibir señales de nuestro entorno y actuar en consecuencia, algunos como la vista y la audición nos permiten detectar y evitar obstáculos o peligros de nuestros alrededores. En el mundo de la robótica, la percepción del entorno es un aspecto que cada vez cobra más importancia pues se hace necesario dotar a los robots de medios para simular dichos sentidos y obtener así una información muy valiosa y esencial para las operaciones autónomas. Para ello, se instalan en los robots una extensa variedad de sensores para medir un gran número de magnitudes físicas como la luz, proximidad, temperatura, sonido, aceleración, etc., otros permiten obtener datos de localización, orientación, imágenes, vídeos... Algunos de estos sensores, como por ejemplo los *LiDARs* y las *cámaras de profundidad*, generan las llamadas nubes de puntos, que consisten en un conjuntos de datos en un espacio 3D que nos permiten representar cualquier objeto o forma y en las cuales se ha basado completamente este proyecto.

Al igual que ocurre con las señales que recibimos de nuestros sentidos, la información recibida de los sensores necesita también ser procesada para saber cómo actuar ante ella. Para procesar las nubes de puntos se hace uso de librerías de datos 3D, que nos permiten analizarlas y modificarlas de forma que, mediante software, podemos hacer que el robot realice diferentes acciones en función de los elementos que haya detectado en su entorno de trabajo.

A día de hoy, hay una infinidad de funcionalidades y algoritmos desarrollados y optimizados para tratar estas nubes de puntos, sin embargo, este tipo de información puede llegar a tener un elevado coste computacional y provocar retrasos en sus operaciones, lo cual puede suponer un gran problema para los sistemas de tiempo real, sobretodo si las operaciones las realizan de forma autónoma. Es por esto que surge la necesidad de hacer uso de la *GPU* para aligerar la carga de la *CPU* y así reducir considerablemente los tiempos de ejecución de las operaciones. En este trabajo se va a realizar una comparativa, en relación a los tiempos de ejecución y el rendimiento obtenido, para una serie de algoritmos tras implementarlos tanto en CPU como en GPU.

1.1 Motivación

En el grado de GIERM se han impartido los conocimientos básicos de la percepción y de la robótica, pero al tratarse de dos materias tan importantes y que están en auge actualmente era realmente interesante ampliar mis conocimientos en estos campos.

Otra de las principales motivaciones para realizar este proyecto ha sido el investigar y trabajar en un área como el procesamiento gráfico en la que no tenía experiencia y en la cual cada vez más empresas de la industria ponen su atención, ya que las ventajas que se obtienen de su uso son muy decisivas.

1.2 Objetivos

El principal objetivo de este proyecto es el desarrollo y optimización de una serie de algoritmos o funcionalidades bastante conocidos mediante el uso de la GPU, para poder así realizar una comparativa de los

resultados obtenidos, en tiempos de ejecución y rendimiento, al ejecutar dichos algoritmos en su versión de CPU y en la de GPU en diferentes escenarios. Esta comparación nos permitirá saber si realmente el uso de GPU en estos algoritmos supone una mejora considerable y también en qué tipo de escenarios es trivial disponer de esta unidad de procesamiento gráfico en los UAVs.

1.3 Estructura del trabajo

Este proyecto podría dividirse principalmente en tres tareas. La primera de ellas sería el estudio y familiarización con diferentes librerías de datos 3D para poder analizar las nubes de puntos de las que disponía, y también con la plataforma *CUDA* para el desarrollo de funcionalidades mediante GPU. La segunda etapa, y la más gruesa, de este trabajo consiste en la elección y desarrollo de las funcionalidades más interesantes para implementarlas con el uso de GPU, buscando en todo momento optimizarlas lo máximo posible. La tercera y última tarea abarca la preparación de escenarios y dispositivos a usar para la obtención de las métricas necesarias para la comparativa y posteriormente sacar tablas y gráficas con dichos resultados.

2 Estado del Arte

En este segundo capítulo se va a analizar la situación actual y las últimas novedades de las nubes de puntos, las librerías de datos 3D y el procesamiento en GPU.

2.1 Nubes de puntos

Como se ha mencionado anteriormente, las nubes de puntos son un conjunto de datos en el espacio 3D que pueden representar objetos, relieves, personas, etc. La enorme cantidad de aplicaciones en la industria que pueden tener los datos 3D está provocando que cada vez más empresas busquen trabajar con éstos. En la rama de la robótica, una de las aplicaciones de los datos 3D que más crecimiento está teniendo y cuyas técnicas están constantemente mejorando es el *mapeado* 3D, que consiste en detectar todos los elementos, dentro de un límite, del entorno en el que se va a operar con el robot y crear un "mapa" con ellos. Un ejemplo de esta técnica puede encontrarse en los LiDARs 3D, que son unos dispositivos que mediante un escaneo láser nos permiten obtener una nube de puntos con los elementos que se encuentren alrededor del sistema. Las ventas de estos dispositivos se están disparando pues se han convertido en soluciones muy útiles para operaciones de detección de obstáculos, o las que necesiten obtener la localización y la odometría en base a dichas nubes de puntos (especial mención al algoritmo *LOAM*[1] y sus variantes *LIOM* y *LIO-SAM*[2]).

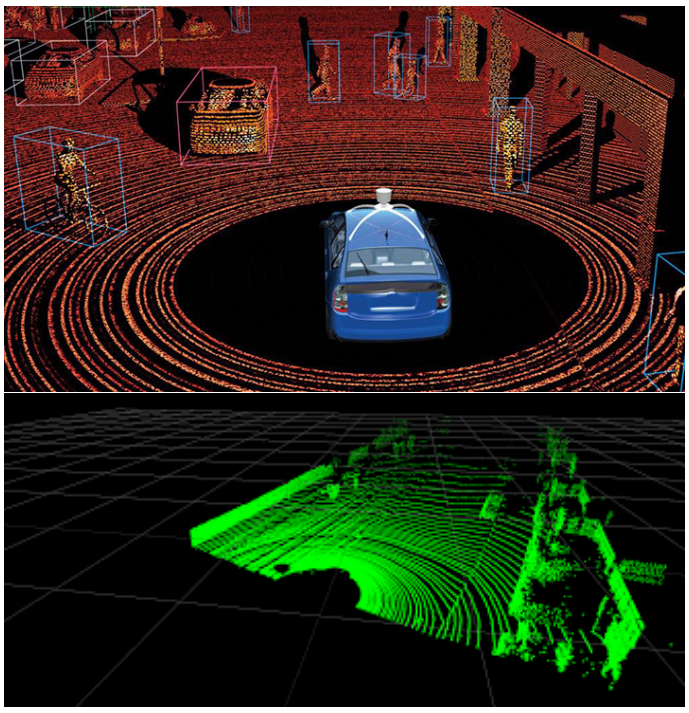

Figura 2.1 Ejemplos de nubes de puntos de un LiDAR-3D.

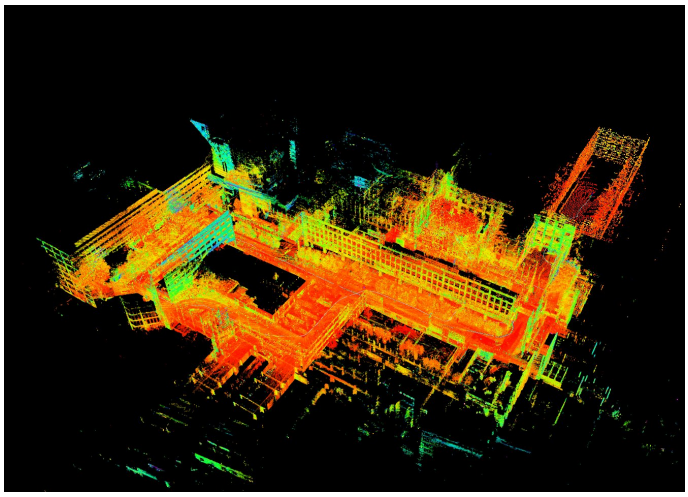

Figura 2.2 Ejemplo de mapeado 3D con el algoritmo LIO-SAM.

2.2 Librerías de datos 3D

Cuando se trata de procesar la información de las nubes de puntos se suele hacer uso de librerías de datos 3D, siendo *PCL* una de las más populares de la industria, sobre todo si trabajas con ROS. Pero, a día de hoy, hay muchas más alternativas y que son más modernas, como por ejemplo *OPEN3D*.

2.2.1 PCL

Point Cloud Library o *PCL* es una librería de código libre enfocada al procesamiento de nubes de puntos e imágenes 2D y 3D. Entre las ventajas de esta librería destacan su característica *multiplataforma*, pudiéndose trabajar en *Linux*, *MacOS*, *Windows* y *Android*; su compatibilidad con *ROS* y su amplio repertorio de algoritmos punteros como los filtrados de puntos, los estimadores de características o *features*, los de reconstrucción de superficies, los de segmentación o los ajustes de modelos. PCL está dividida en varias librerías más pequeñas o módulos de compilación independiente, algunos ejemplos de módulos son los filtros, la segmentación y la búsqueda de vecinos.

Figura 2.3 Logo PCL.

Figura 2.4 Ejemplo del módulo Registration de PCL.

El conjunto de todos estos módulos permiten numerosas aplicaciones como la eliminación de ruido, la segmentación o división de una escena en sus partes más interesantes, la extracción de puntos característicos, la detección de objetos por su forma geométrica o la creación y visualización de superficies a partir de nubes de puntos. Desde el lanzamiento de su primera versión oficial en 2011, esta librería ha continuado creciendo y aumentando su catálogo de algoritmos, convirtiéndose así en una de las librerías más usadas en la comunidad robótica para el procesamiento de datos 3D. PCL fue diseñada en un principio[3] con el objetivo de enfocarse en la eficiencia y rendimiento en las CPUs modernas con el apoyo de *OpenMP* para la paralelización de los procesos en varios núcleos, la librería *FLANN* para unas rápidas búsquedas de vecinos y el uso de punteros compartidos de *Boost* para evitar múltiples copias innecesarias de la información y actualmente cuenta también con soporte para las GPUs mediante el uso de *CUDA* y *OPENCL*.

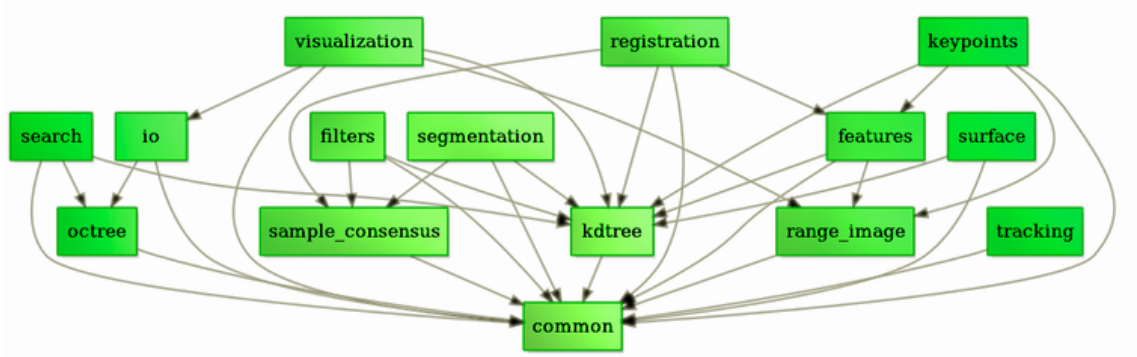

Figura 2.5 Módulos de PCL.

2.2.2 Open3D

Open3D[4] es otra librería de código abierto y, al igual que *PCL*, está enfocada en el desarrollo de software para trabajar con datos 3D. Es una librería muy moderna, su primera versión oficial salió en 2018, que parte de una base altamente optimizada y preparada para la paralelización y además cuenta con una gran variedad de funcionalidades punteras en su catálogo. Al ser una librería en pleno crecimiento hay algoritmos o funcionalidades que pueden resultar útiles para el procesamiento de nubes de puntos que aún no están implementadas, sin embargo, los colaboradores de *OPEN3D* trabajan de la mano de algunos colaboradores de *PCL* para incluirlas. Esta librería multiplataforma permite la compatibilidad con *ROS*, *OpenMP*, *CUDA*, *FLANN* y *Eigen* entre otros, sus desarrollos están implementados tanto en *C++* como en *Python* y cuenta con un gran número de colaboradores para mejorar la librería día tras día con la ayuda de la comunidad detectando errores y posibles mejoras. Entre su documentación se encuentran una infinidad de tutoriales y ejemplos que te permiten familiarizarte con este entorno moderno.

Figura 2.6 Logo OPEN3D.

Esta librería también está dividida en una serie de módulos clasificando así cada una de sus funcionalidades según su categoría. Entre ellos se pueden destacar el módulo *ml* que alberga una extensión de Open3D para las tareas de *machine learning*; el módulo *geometry* que contiene una gran variedad de funcionalidades para el procesamiento de nubes de puntos; el módulo *visualization* que permite visualizar con gran detalle las nubes de puntos con las que se está trabajando; el módulo *core* que permite entre otras cosas el uso de algoritmos de búsqueda de vecinos, las conversiones con datos *Eigen* y el uso de una estructura de almacenamiento de datos llamada *Tensor* y, por último, el módulo más importante para este proyecto, el módulo *t* que contiene muchas de las anteriores funcionalidades implementadas para trabajar con datos almacenados en los ya mencionados tensores. Este tipo de datos ya se analizará con más detalle en los siguientes capítulos, pero básicamente estas estructuras nos permiten almacenar los datos tanto en la *CPU* como en la *GPU*.

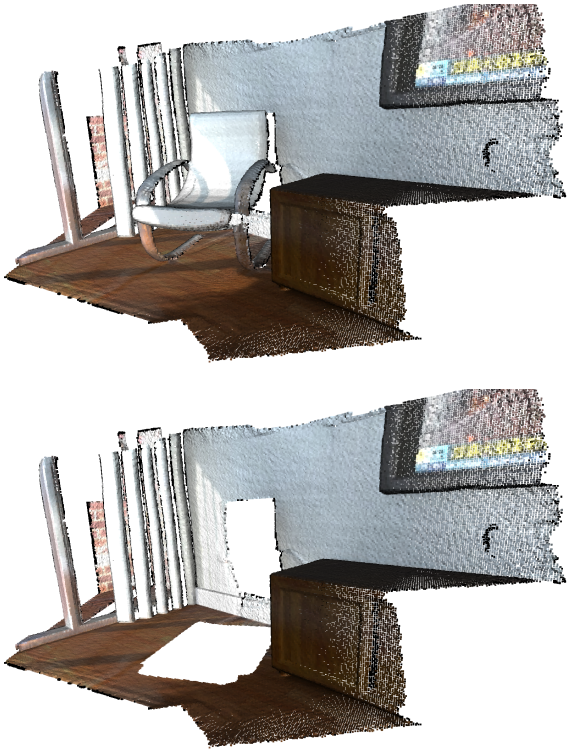

Figura 2.7 Ejemplo de manipulación de nubes de puntos del módulo Geometry de OPEN3D.

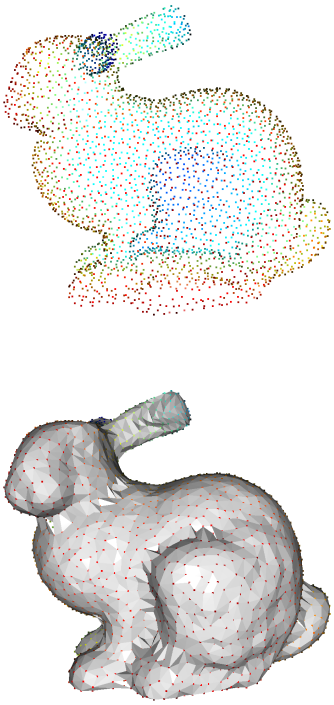

Figura 2.8 Ejemplo de reconstrucción de superficies del módulo Geometry de OPEN3D.

2.3 Procesamiento en GPU

Una unidad de procesamiento gráfico o *GPU* es un coprocesador diseñado especialmente para el procesamiento en paralelo y operaciones de coma flotante, permitiendo así aligerar la carga de trabajo de la CPU en tareas de alto coste computacional[5]. La principal diferencia existente entre una CPU y una GPU reside en el diseño de sus arquitecturas, el de la CPU está enfocado en la latencia, es decir, en ejecutar lo más rápido posible las tareas complejas y esto lo consigue mediante el uso de varios hilos software en un procesamiento en serie secuencial manejados por unos cuantos núcleos existentes en la CPU; mientras que la GPU está diseñada para el rendimiento mediante la paralelización de los múltiples hilos que pueden manejar simultáneamente los miles de núcleos de la GPU (más pequeños que los de la CPU). Debido a sus características, la GPU era, en sus inicios, comúnmente usada para tareas de procesamiento de imágenes o vídeos y videojuegos, y en la actualidad resultan de una enorme utilidad para la *inteligencia artificial*, el *deep learning*[6] o el minado de *criptomonedas*[7] gracias a sus avanzadas y mejoradas arquitecturas.

Desde el lanzamiento de las primeras GPUs, entre finales del siglo XX y los inicios del siglo XXI, las arquitecturas de las GPUs no han parado de mejorar, se ha demostrado que desde el lanzamiento de la plataforma *CUDA* allá por 2007, el crecimiento de las GPUs supera el ritmo de la *Ley de Moore* llegando a triplicar los rendimientos cada 2 años[8]. Existen varios fabricantes de GPU en el mercado siendo los más conocidos *AMD*, *ATI*, *Intel* y especialmente *NVIDIA*, creador de la ya mencionada plataforma CUDA de la mano de las series de GPUs *GeForce 8M* con la micro-arquitectura *Tesla*. Actualmente, las GPUs más recientes de este fabricante pertenecen a las series *GeForce 30* y *MX* con la micro-arquitectura *Ampere* para versiones de escritorio. NVIDIA también cuenta con un sistema en chip llamado *Tegra*[9] para dispositivos móviles y sistemas embebidos del mismo fabricante como las placas *Jetson TX1/2*, *Jetson Nano*, *Jetson Xavier* o la más reciente *Clara Holoscan* con micro-arquitecturas *Maxwell*, *Pascal*, *Volta* y *Ampere*[10]. En este proyecto, aprovechando que puede embarcarse en un *UAV*, se usará la placa Jetson TX2 para la obtención de resultados.

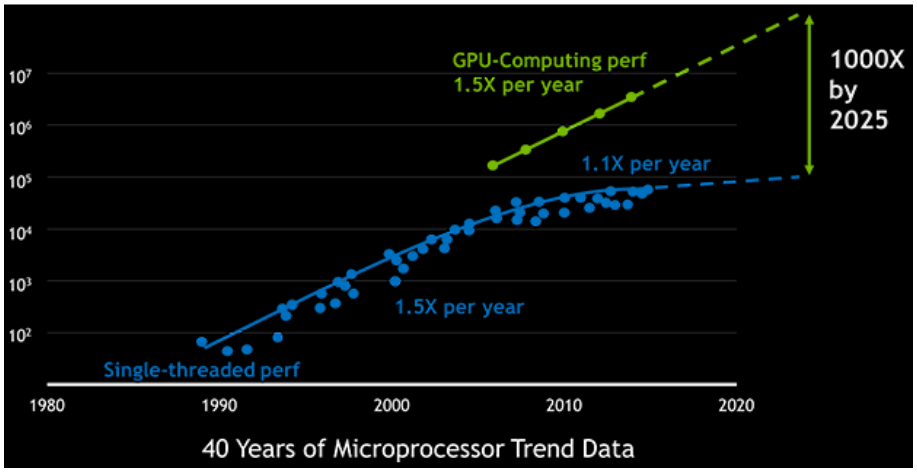

Figura 2.9 Comparación del crecimiento de las GPUs con la Ley de Moore.

Debido al auge y crecimiento de las GPUs en la industria tecnológica, las librerías y algoritmos más punteros, en cuanto a procesamiento de gráficos e inteligencia artificial se refiere, cuentan hoy en día con soporte para implementaciones en la GPU con el objetivo de obtener así grandes rendimientos.

Figura 2.10 Kit de desarrollo de una NVIDIA Jetson TX2.

3 Procesamiento de nubes de puntos con GPU

Las librerías de procesamiento de datos 3D se apoyan en el uso de CUDA para implementar a bajo nivel algunos algoritmos, de forma que puedan usar la GPU para obtener resultados más rápidos y eficientes. Para este proyecto en el que usamos C++ como lenguaje de programación y trabajamos en el entorno de *ROS* habrá que encontrar la forma de que todo el código del proyecto compile sin problemas para generar el ejecutable haciendo uso de la herramienta *CMake*.

En este capítulo se analizarán también los algoritmos de procesamiento de nubes de puntos presentes en las librerías mencionadas en el Capítulo 2 y se describirán detalladamente las funcionalidades que se han elegido desarrollar en GPU para este proyecto.

3.1 Entorno de programación

Como ya se ha comentado, para poder implementar funcionalidades que hagan uso de la *GPU*, estos algoritmos deberán estar programados a bajo nivel mediante *kernels* de *CUDA*. CUDA, que proviene de *Compute Unified Device Architecture*, consiste en una plataforma de computación en paralelo desarrollada por *NVIDIA* que permite a los programadores implementar algoritmos en GPUs del fabricante NVIDIA.

En la programación en CUDA las funciones que se ejecutan en el contexto de la GPU vienen precedidas del especificador *__global__* y son conocidas como *kernels*, estos son llamados para ejecutarse mediante el uso de los llamados *kernel launch*, fácilmente distinguibles por sus triples comillas angulares (<<<...>>>). Como la CPU y la GPU poseen su propio espacio de memoria, la CPU no puede acceder directamente a la memoria de la GPU y viceversa. El procedimiento para trabajar con programas en CUDA e intercambiar datos entre la CPU y la GPU consiste en reservar memoria de la CPU e inicializar sus datos, luego reservar memoria en la GPU y transferir los datos de la memoria CPU a la de la GPU, ejecutar el kernel deseado y finalmente transferir de vuelta los datos obtenidos desde la memoria de la GPU hasta la de la CPU. La gestión de la memoria en la GPU es similar a la de la CPU, en vez de usar las funciones *malloc()*, *free()* y *memcpy()* se usarán sus equivalentes *cudaMalloc()*, *cudaFree()* y *cudaMemcpy()*. Actualmente, las librerías de datos 3D proporcionan suficientes funciones de alto nivel como para que no tengamos que preocuparnos por la gestión de memoria entre ambas entidades.

La llamada a un kernel funciona prácticamente igual que las llamadas a funciones de C con la única diferencia de la adición de dos parámetros entre las comillas angulares. El primero de ellos indica la cantidad de bloques de hilos que se lanzarán, mientras que el segundo indica la cantidad de hilos paralelos que conforman un bloque de hilos. Para aprovechar al máximo la paralelización de la GPU, se suele hacer uso de varios bloques con un número considerable de hilos de forma que se pueda asignar un hilo a cada uno de los elementos que haya que analizar [11] como puede verse en la Figura 3.1. Al hacer uso de la paralelización de hilos hay que tener especial cuidado con la memoria compartida pues pueden producirse condiciones de carrera si no se garantiza el acceso exclusivo de un hilo a una variable compartida[12].

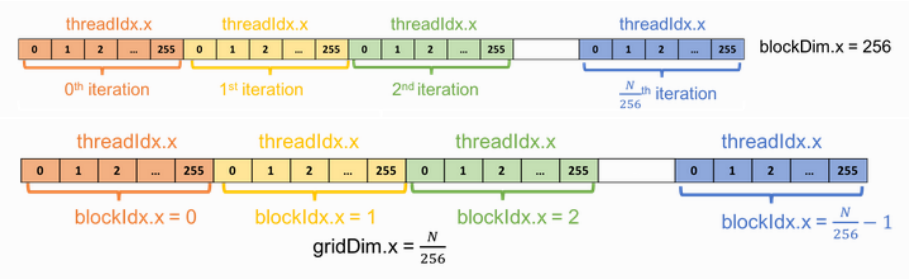

Figura 3.1 Ilustración de la paralelización de una función con 1 y varios bloques respectivamente.

Para que el código se ejecute en la GPU éste tiene que estar contenido en un archivo con extensión *".cu"* para que el compilador de CUDA llamado *nvcc* lo detecte y lo compile separadamente con respecto al código de CPU. Cuando se trabaja con ROS es habitual compilar los códigos fuente mediante la herramienta de *CMake*, por lo que será necesario detectar el paquete de *CUDAToolkit* [13] y activar una serie de "flags" en el archivo *"CMakeLists.txt"* que permitan realizar una *compilación separada* en un proyecto que contenga a la misma vez archivos de C/C++ y CUDA[14].

3.2 Algoritmos de procesamiento

De la librería PCL ya se ha comentado que tiene un catálogo de funcionalidades bastante amplio [15] pero para este apartado se van a nombrar los algoritmos más útiles o más usados en aplicaciones de procesamiento de nubes de puntos en sistemas autónomos. Los más destacables podrían ser:

- El *extractor euclidiano de clusters* (pcl::extractEuclideanClusters) que sirve para segmentar la nube de puntos de entrada en grupos de puntos según la distancia entre puntos.
- El *filtro Frustum* (pcl::FrustumCulling) que permite filtrar los elementos de una nube de puntos que se encuentren dentro de un campo de visión determinado.
- El *extractor de outliers por vecindad o estadística* (pcl::RadiusOutlierRemoval y pcl::StatisticalOutlierRemoval) que detecta y elimina los outliers de una nube de puntos de entrada.
- El *filtro Passthrough* (pcl::Passthrough) o el *extractor condicional* (pcl::ConditionalRemoval) que filtran todos los puntos cuyas características no cumplan unos determinados límites o condiciones.
- El *filtro por cuadrículas* (pcl::VoxelGrid) que agrupa varios puntos en cuadrículas 3D, aproximadas por el centroide de los puntos que la conforman, y consigue reducir la resolución o tamaño de la nube de puntos.
- El *extractor de índices* (pcl::ExtractIndices) que filtra una nube de puntos manteniendo los elementos cuyos índices se encuentren en una lista de índices dada.
- El *filtro de caja* (pcl::CropBox) que filtra una nube de puntos manteniendo los elementos que se encuentren dentro de los límites de un cubo determinado.
- La *segmentación SAC* (pcl::SACSegmentation) que segmenta la nube de puntos mediante métodos y modelos basados en el *Sample Consensus*. Permite obtener los elementos de la nube de puntos que, por ejemplo, mejor se ajusten a un plano o un cilindro.
- Los *Kd-tree* (pcl::KdTree) son unas estructuras de datos que permiten unas búsquedas rápidas de los vecinos más cercanos a un punto.

Como puede observarse, la mayoría de las funcionalidades destacadas solo tienen implementación en CPU, siendo el *extractor euclidiano de clusters*, el *extractor de índices* y la *segmentación SAC* los únicos de los mencionados arriba que tienen implementación en GPU.

La librería *Open3D* contiene la gran mayoría de los algoritmos [16] que se han destacado de PCL (open3d::geometry::KDTreeFlann, open3d::geometry::VoxelGrid, open3d::geometry::PointCloud::Crop, open3d::geometry::PointCloud::SegmentPlane y open3d::geometry::PointCloud::SelectByIndex). Sin embargo, de todos estos solo el *KDTreeFlann* tiene implementación en CUDA.

3.3 Algoritmos seleccionados

3.3.1 Filtro Frustum

Este tipo de filtro consiste en identificar los puntos de la nube de puntos que se encuentran dentro de un campo de visión dado por una serie de parámetros compuesta por las distancias a los planos más cercano y más lejano a la cámara, la orientación de la cámara y los ángulos de visión vertical y horizontal. Mediante estos parámetros se pueden calcular las ecuaciones de los planos laterales, el superior y el inferior, los cuales delimitan junto a los planos cercano y lejano el campo de visión deseado. Con la obtención de estas ecuaciones, al filtro solo le queda evaluar uno por uno todos los puntos de la nube de puntos para comprobar si se encuentran dentro o fuera del campo de visión y almacenar los índices de los que cumplan la condición.

Esta funcionalidad nos sirve como una forma de preprocesar la nube de puntos que obtenemos y centrarnos exclusivamente en un campo de visión determinado, reduciendo considerablemente la carga computacional de los siguientes algoritmos que se quieran aplicar. Este filtro es realmente útil para la detección de obstáculos ya que nos permite ignorar aquellos elementos de la nube de puntos que se encuentren fuera del campo de visión deseado y así poder trabajar íntegramente con obstáculos que deban ser tenidos en cuenta según su ubicación en el entorno del UAV.

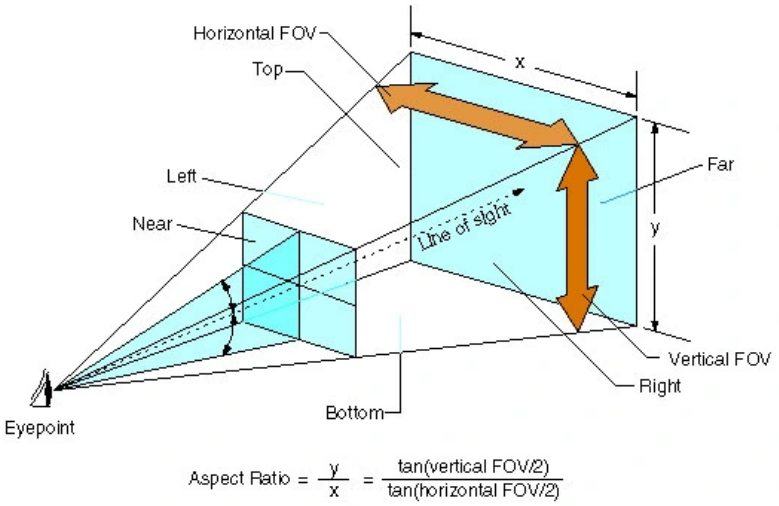

Figura 3.2 Ilustración del campo de visión en un filtro frustum.

En la Figura 3.3 puede observarse cómo la aplicación de este filtro nos permite identificar los obstáculos que se encuentren enfrente del UAV y los que podrían encontrarse debajo mediante un segundo *frustum*. Este ejemplo está sacado de la detección de obstáculos de un proyecto realizado por la empresa FADA-CATEC.

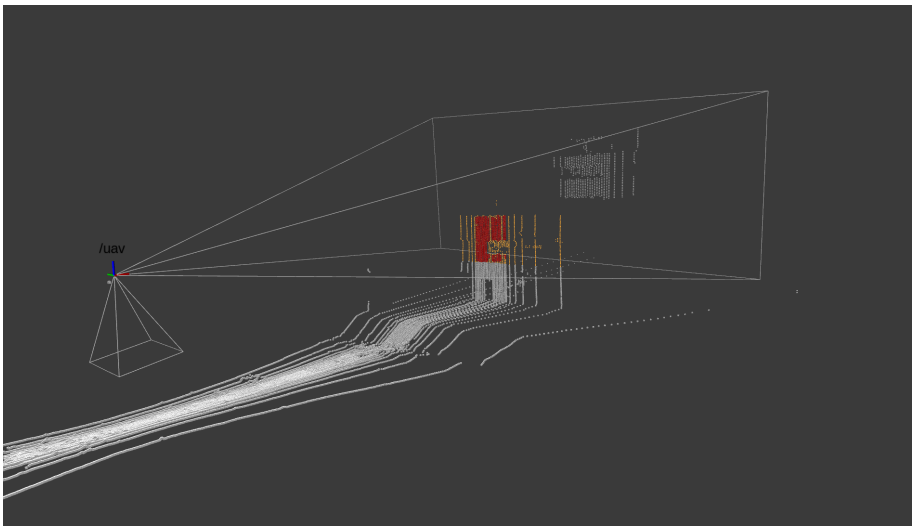

Figura 3.3 Ejemplo de aplicación de un filtro frustum en nubes de puntos.

3.3.2 Extractor de índices

El *extractor de índices* es una funcionalidad básica cuyo funcionamiento está basado en filtrar una nube de puntos de entrada de forma que los puntos de ésta cuyos índices se encuentren en una *lista de índices*, que se le pasa como parámetro al algoritmo, serán mantenidos en la nube de puntos de salida mientras que el resto son excluidos de dicha nube de puntos resultante.

Esta funcionalidad sirve de base para muchos de los algoritmos de procesamiento de nubes de puntos ya que, a partir de una lista de índices, que puede generarse por otro algoritmo, podemos obtener una nube de puntos exclusivamente compuesta con los elementos de la nube que nos interesen o cumplan una determinada condición. Dentro de los algoritmos que he seleccionado, el *filtro frustum* y el *extractor de puntos más cercanos* proporcionan una lista de índices con la que se puede aplicar este extractor de índices, mientras que el *extractor de outliers* directamente incluye este algoritmo dentro de su funcionamiento.

3.3.3 Extractor de outliers

Esta funcionalidad consiste en analizar la nube de puntos y clasificar sus elementos en *inliers* y *outliers*. Para realizar esa clasificación, el algoritmo analiza cada uno de los puntos de la nube y busca un cierto número de vecinos alrededor del punto analizado en un determinado radio de vecindad, tanto el número de vecinos como el radio de vecindad son parámetros de entrada del algoritmo. Si se encuentran los suficientes vecinos el punto analizado se considera inlier, y si no sería un outlier. Una vez que se han clasificado todos los puntos en inliers y outliers, se aplica el extractor de índices para obtener una nube de puntos que excluya a los outliers detectados.

En la Figura 3.4, se observa el análisis de la vecindad de 3 puntos. Si suponemos que necesitamos como mínimo 2 vecinos dentro de un radio de vecindad d para que un punto sea considerado un inlier, tanto el punto de la izquierda como el de la derecha no cumplen dicha condición al disponer de 0 y 1 vecinos en su entorno, respectivamente, y serán considerados outliers; mientras que el punto del centro cuenta con 4 vecinos en su entorno de vecindad por lo que será considerado un inlier.

Esta funcionalidad es de gran utilidad cuando se dispone de nubes de puntos con un nivel de ruido considerable, ya que el algoritmo nos eliminaría gran parte de éste, evitando así que el ruido afecte negativamente al funcionamiento de la aplicación que uno quiera desarrollar. Normalmente, las nubes de puntos obtenidas por un *LiDAR* contienen algunos puntos que no corresponden realmente con superficies físicas y son considerados outliers, por lo que éstos deberían ser eliminados antes de seguir procesando la nube de puntos.

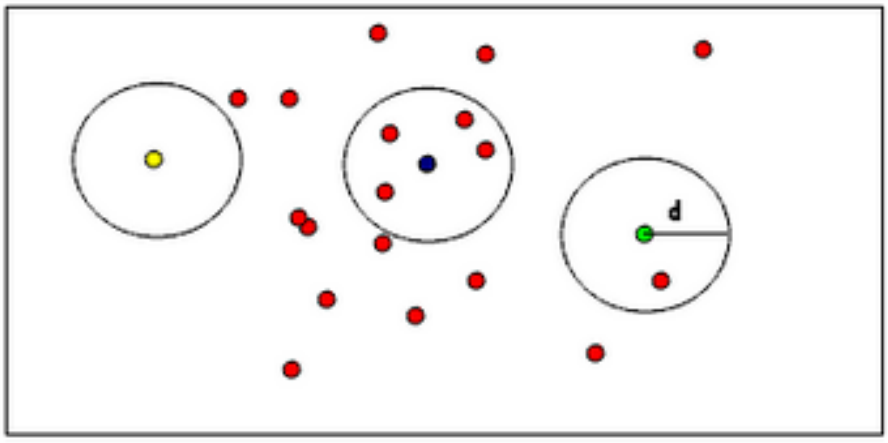

Figura 3.4 Ilustración de la extracción de outliers por radio de vecindad.

3.3.4 Extractor de puntos más cercanos

El funcionamiento de este algoritmo se basa en calcular la distancia de todos los elementos de una nube de puntos con respecto a un punto de referencia para luego poder identificar aquellos puntos que se encuentren más cerca de esta referencia. Para asegurarse de que el punto no se trata de una medidad ruidosa o de un elemento minúsculo se puede elegir aplicar una búsqueda de vecindad para identificar también aquellos puntos que sean vecinos de los puntos más cercanos, parecido a lo que se realiza en la obtención de *clusters*. Esta funcionalidad permitiría detectar los obstáculos más cercanos a un UAV para poder así actuar en consecuencia y evitarlos. Tanto en este algoritmo como en el *extractor de outliers* se requiere de un análisis de vecindad, lo cual aumenta considerablemente el coste computacional y sus implementaciones en GPU pueden resultar de gran interés.

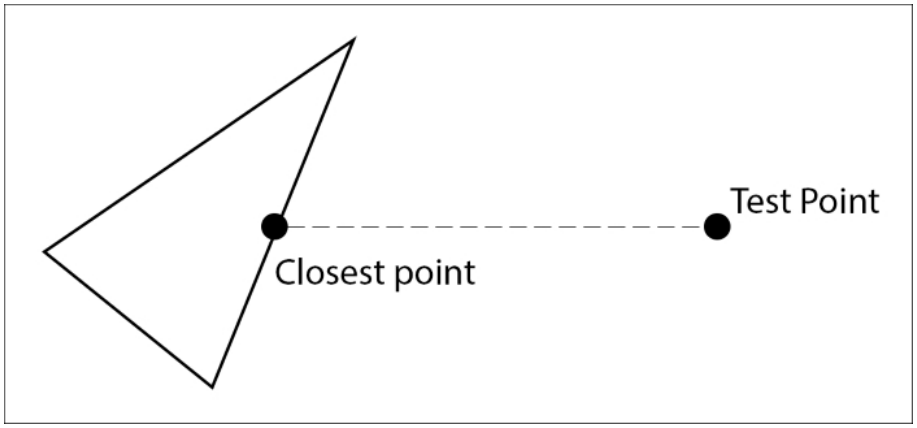

Figura 3.5 Ilustración de punto más cercano a una forma geométrica.

3.3.5 Creador de nube de puntos aleatorios

Esta funcionalidad consiste, como su nombre indica, en crear una nube de puntos formada por puntos cuyas coordenadas sigan una distribución aleatoria. Los valores límite de las coordenadas de dichos puntos aleatorios y el tamaño de la nube aleatoria generada serán recibidos como parámetros de entrada del algoritmo. Esta función nos servirá en nuestro proyecto para incluir un número de puntos ruidosos conocido a otra nube de puntos y evaluar así la eficacia del *extractor de outliers* desarrollado.

Figura 3.6 Ejemplo de nube de puntos aleatorios.

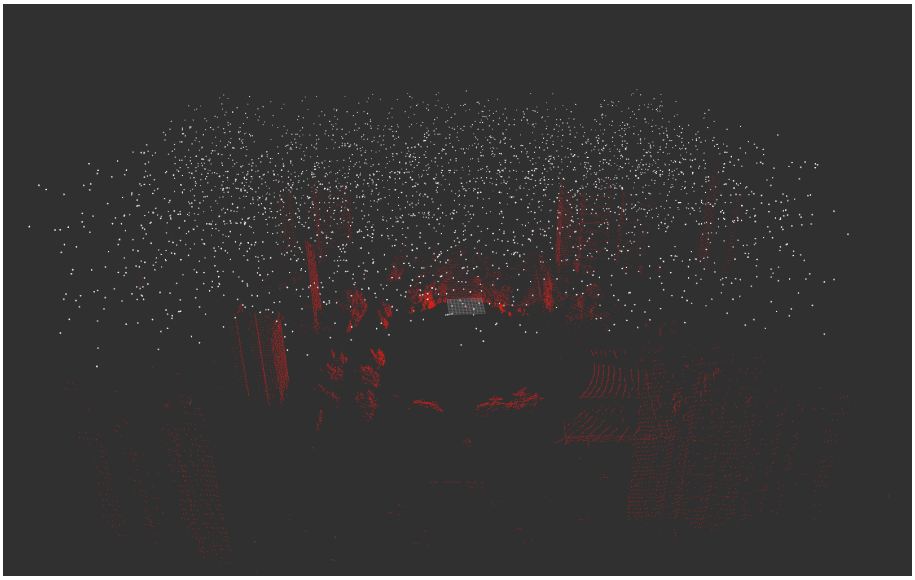

Figura 3.7 Ejemplo de nube de puntos aleatorios en el entorno de otra obtenida mediante un LiDAR.

3.4 Conclusión

Debido a una serie de errores bloqueantes en la ejecución de algoritmos de PCL con CUDA se optó por usar la librería Open3D y se estudiaron los algoritmos que fuesen más interesantes para desarrollar su implementación en CUDA y que no estuviesen ya incluidos en la librería. Los algoritmos seleccionados se escogieron por sus utilidades en las operaciones de los sistemas autónomos y, por tanto, por las posibles ventajas en términos de rendimiento y tiempos de ejecución que supondría usar la GPU para estas operaciones.

4 Elección y preparación del dataset

Una parte fundamental de este proyecto ha sido la elección y preparación de un dataset que me permitiese analizar el funcionamiento de los algoritmos desarrollados en una serie de escenarios característicos. Para comprobar dicho funcionamiento se han usado sets de datos reales capturados en entornos urbanos con un LIDAR 3D embarcado en un UAV. Concretamente se han usado unos rosbags proporcionados por la empresa FADA-CATEC que fueron grabados en un parque en mitad de la ciudad de Benidorm para el proyecto DELOREAN. Las pruebas de vuelo en las que se grabaron estos sets de datos tenían la finalidad de obtener información para trabajar con la localización y generación de mapas en entornos urbanos. En dicho dataset los principales obstáculos que se detectan son los edificios de alrededor y los árboles y palmeras del parque, de forma que según la altura en la que se encuentre volando el dron y el desplazamiento en su entorno variará considerablemente el número de obstáculos detectados.

4.1 Dataset

De cara a comparar los resultados obtenidos de la ejecución de los algoritmos desarrollados mediante CPU y GPU, se creó la necesidad de buscar dentro del set de datos ya mencionado los escenarios más característicos. Tras varias ejecuciones de los algoritmos se pudo determinar que los escenarios más interesantes para mostrar los resultados serían los siguientes.

4.1.1 Dron en el suelo

Un primer escenario interesante es aquel en el que el dron se encuentra en el suelo (ya sea antes de despegar o una vez ya aterrizado) ya que en esta situación se obtiene una nube de puntos bastante densa mediante el LIDAR que conlleva a una gran carga computacional en los algoritmos desarrollados que incluyen búsquedas de vecinos y, en consecuencia, se obtienen unas métricas de tiempo bastante grandes en comparación al funcionamiento deseado. En la Figura 4.1 se muestra un instante del rosbag escogido para este escenario, en el que puede comprobarse la gran densidad de la nube de puntos en el suelo y las palmeras cercanas.

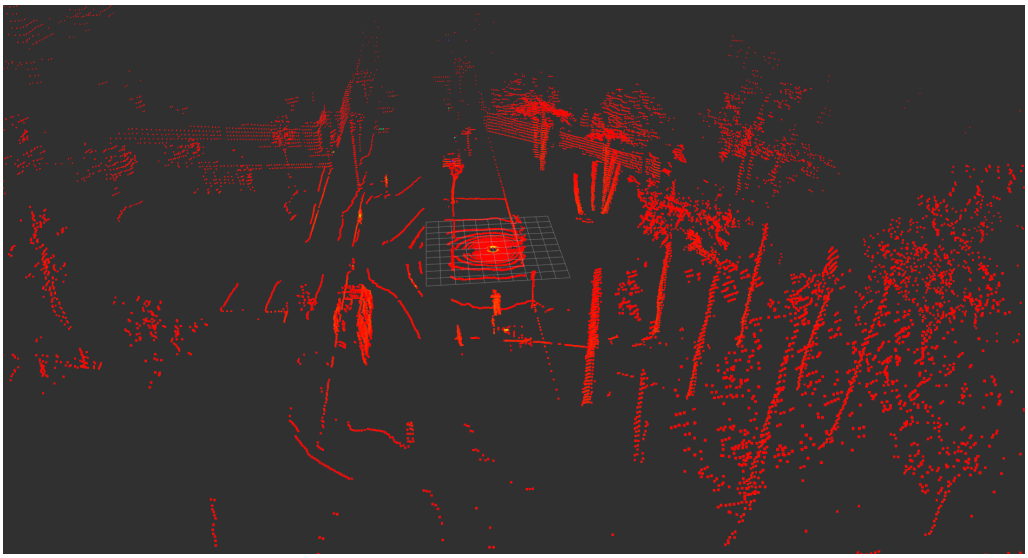

Figura 4.1 Escenario dron en el suelo.

4.1.2 Aterrizaje

El segundo escenario a analizar será el proceso de aterrizaje (también aplicable al proceso de despegue) en el cual se obtienen unas nubes de puntos mucho más grandes que las obtenidas cuando el dron se encuentra volando a unas mayores alturas, pero no serán tan densas como las obtenidas en el primer escenario descrito. Este escenario es bastante interesante pues el momento de aterrizaje es una situación clave a tener en cuenta en todo vuelo que se realice en el que se quiera tener en cuenta la detección de obstáculos. En la Figura 4.2 puede verse un instante de dicho escenario en el que se detecta una gran cantidad de puntos de suelo y de las palmeras de alrededor.

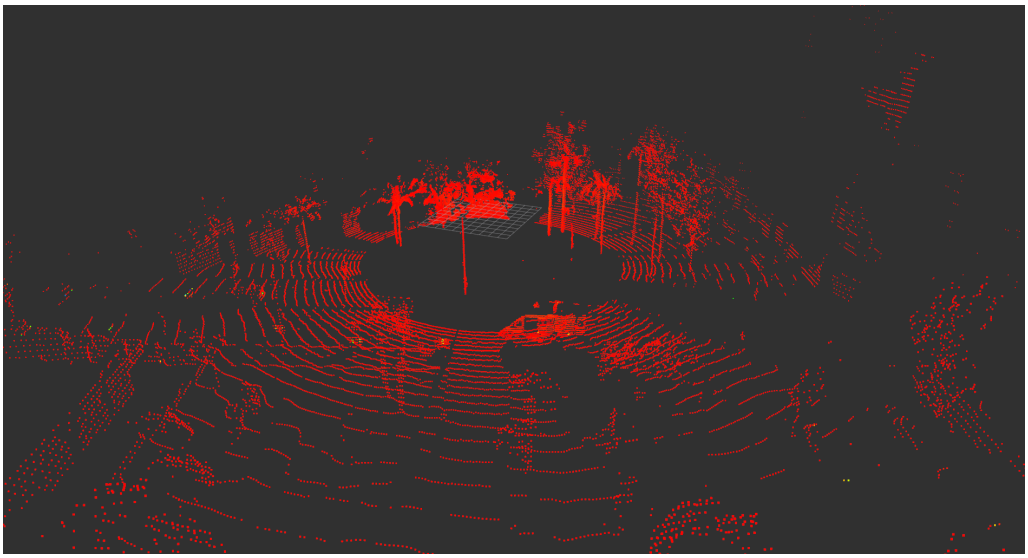

Figura 4.2 Escenario aterrizaje.

4.1.3 Nube de puntos pequeña

Otro escenario que se ha escogido ha sido aquel en el que el dron está volando a una altura ya considerable y no se detectan apenas los árboles y palmeras (sus copas como mucho) pero sí que se siguen detectando algunas fachadas de los edificios que están un poco alejados, dando lugar a la obtención de nubes de puntos pequeñas de tamaño y relativamente dispersas con respecto a las obtenidas en otras situaciones. Esto se ve reflejado en la Figura 4.3, en la que el dron está avanzando y si se le aplicase un frustum se perdería de vista el edificio de la izquierda y solo se obtendrían los conjuntos de puntos poco densos correspondientes a las palmeras y al edificio del fondo.

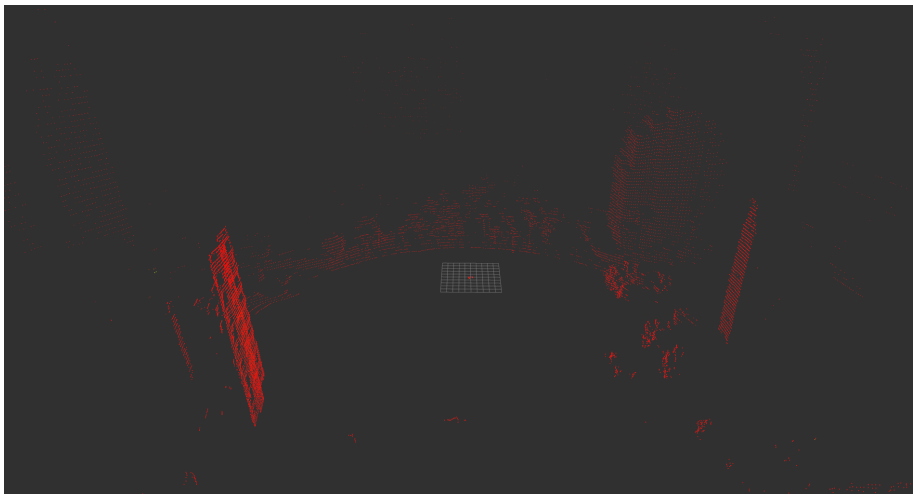

Figura 4.3 Escenario nube pequeña.

4.1.4 Nube de puntos intermedia

Un último escenario que se ha considerado interesante para analizar ha sido aquel en el que el dron se encuentra en una situación parecida a la del tercer escenario pero con la diferencia de que en este caso sí se detectan mejor los árboles/palmeras al estar volando algo más bajo y los edificios están un poco más cerca, obteniendo así una nube de puntos de tamaño y densidad intermedias. Esto puede verse en la Figura 4.4, en la que en el instante mostrado, el dron está avanzando hacia una zona en la que tiene cerca palmeras y edificios que entrarían en el campo de visión si se le aplicase un filtro frustum. Estos conjuntos de puntos son algo más densos que los citados en el anterior escenario.

Figura 4.4 Escenario nube intermedia.

4.2 Conclusión

Es obvio llegar a la conclusión, de que las densas nubes de puntos del primer escenario (dron en el suelo) o las grandes del segundo (dron aterrizando) hacen visible la necesidad de un aceleramiento gráfico que reduzca en gran medida los tiempos de ejecución en estos momentos clave. Los otros dos escenarios sirven para demostrar los límites en los que el aceleramiento gráfico y su correspondiente gestión de memoria sigan siendo viables, pues habrá situaciones en las que la carga computacional sea tan baja que se obtendrán mejores rendimientos en CPU que en GPU. En definitiva, la elección de estos cuatro escenarios nos permitirá distinguir, tras analizar posteriormente las métricas obtenidas, las situaciones que requieran una ejecución con la GPU de las que no lo requieran.

5 Desarrollo de los algoritmos

El grueso de este proyecto recae en el desarrollo de los algoritmos seleccionados que se mencionaron en el capítulo 3. En este quinto capítulo se van a tratar con detalles los procesos de diseño e implementación de dichos algoritmos.

Con vistas a la obtención de resultados y su posterior comparativa, se vio la necesidad de desarrollar una nueva funcionalidad aparte de los algoritmos ya mencionados anteriormente. Se trata de la creación de una nube de puntos aleatorios que nos permitiese representar unos outliers conocidos y así obtener un porcentaje de outliers eliminados al juntarla con la nube de puntos principal con la que se trabaja y luego aplicarle el procesamiento correspondiente.

5.1 Diseño

En este apartado se va explicar brevemente en qué consiste o qué funcionamiento se busca para cada uno de los algoritmos.

5.1.1 Filtro Frustum

Tal y como se ha comentado en la Sección 3.3, el filtro Frustum es aquel que se basa en un campo de visión de 3 dimensiones determinado por las ecuaciones matemáticas de 6 planos para determinar si un punto se encuentra en dicho campo de visión o fuera de éste.

A la hora de diseñar este filtro, la idea era mantener la estructura de la algoritmia presentada en la librería PCL pero adaptándola a OPEN3D. De esta forma se ha buscado crear un filtro que, a partir de los parámetros de la posición de la cámara, los ángulos de visión horizontal y vertical y las distancias a los planos cercano y lejano, determinase un campo de visión y nos permitiese obtener los puntos de una nube de entrada que se encontrasen en dicho campo de visión, deshaciéndonos así de los que se encuentren fuera de éste. Esta funcionalidad no fue pensada para eliminar los puntos de la nube de entrada que no se encontrasen dentro del campo de visión sino que más bien nos proporcionará los índices dentro de la nube de puntos de los elementos que sí se encuentren en el interior del campo de visión. Si el usuario quisiese eliminar los puntos exteriores al campo de visión a partir de los índices obtenidos tendrá que hacer uso de otro de los algoritmos que se han realizado, el extractor de índices, cuyo diseño se abordará a continuación.

5.1.2 Extractor de índices

Esta funcionalidad es una de las más básicas, está diseñada para recibir como parámetros de entrada un tensor booleano que indique los elementos de la nube de puntos que deben ser mantenidos y los que deben ser eliminados y una variable booleana que sirva para indicar si se quiere o no invertir el valor de dicho tensor provocando así que los elementos que estaban indicados para ser eliminados ahora se mantengan y viceversa. El diseño de esta funcionalidad ha sido basado en el pull-request 3422 de Open3D[17], pues en el momento de la realización de este proyecto dicha funcionalidad aún no había sido implementada con tensores en la librería Open3D. En dicho diseño se hace uso de funciones de alto nivel para una mayor eficiencia en el acceso a los múltiples elementos de las nubes de puntos.

5.1.3 Extractor de outliers

Para el extractor de outliers se ha buscado diseñar un algoritmo que analice la nube de puntos de entrada para detectar los puntos que puedan considerarse outliers, según unos parámetros condicionantes que se usarán en la búsqueda de vecinos, y extraerlos para obtener así a la salida una nube de puntos compuesta por inliers. Dicho algoritmo ha sido basado también en el pull-request 3422, al que se le han añadido y modificado un par de detalles para mejorar su rendimiento en cuanto a tiempos de ejecución.

5.1.4 Extractor de puntos más cercanos

Para esta funcionalidad se han diseñado dos versiones, una que trabajase con tensores y otra sin ellos, pero ambas compartiendo la misma idea de funcionamiento y estructura. Para la versión con tensores se ha necesitado crear una función auxiliar que calculase la distancia de un punto en concreto con el resto de la nube de puntos de entrada, y posteriormente en la función principal se buscarán los valores mínimos entre estas distancias para obtener así los puntos más cercanos al punto especificado. Si se indica mediante un parámetro, se incluirán también los vecinos de los puntos más cercanos para permitir al usuario determinar si se encuentra ante un objeto a tener en cuenta o si son puntos aislados que no interfieran mucho. Por otro lado, para la versión sin tensores, la única diferencia en cuanto a estructura del algoritmo es que no hace falta crear una función auxiliar para calcular las distancias entre un punto y una nube de puntos pues ya existe en la librería de OPEN3D.

5.1.5 Creador de nube de puntos aleatoria

Como ya se ha comentado en la introducción de este capítulo, esta funcionalidad tiene como objetivo crear una nube de puntos aleatoria para evaluar el rendimiento del extractor de outliers. Se le especificará el número de puntos deseados y los valores máximos y mínimos en los 3 ejes cartesianos que deberán tener los puntos aleatorios. Para esta funcionalidad, al igual que con la anterior, también se han diseñado dos versiones, una usando tensores y otra sin ellos, ambas con el mismo principio de funcionamiento. Cada versión constará de una función principal que se encargará de crear la nube de puntos a partir de los puntos aleatorios obtenidos de otra función auxiliar encargada de calcularlos y almacenarlos.

5.2 Implementación

En este segundo apartado, se va a entrar mucho más en detalle con una explicación técnica del código realizado, además de una explicación sobre su preparación y ejecución desde un nodo de ROS.

5.2.1 Filtro Frustum

Para el desarrollo de este filtro se ha creado una clase llamada *FrustumCulling* en cuyo constructor se le debe de pasar la nube de puntos que hay que procesar y la almacena en una variable privada. En dicha clase se han creado una serie de funciones para permitir al usuario establecer los valores de todos los parámetros del filtro según sus requerimientos y otra encargada de aplicar el filtro en cuestión en CPU o en GPU según se le indique.

En primer lugar está *setCameraPose*, la cual nos permite establecer la posición y orientación de la cámara con respecto al origen mediante una matriz 4x4 flotante de la librería *Eigen* cuyas columnas representan el eje de visión de la cámara, el eje vertical, el eje lateral y el vector de traslación. La variable interna que almacena esta matriz tiene que ser inicializada sin alineamiento para evitar un *Eigen Assertion Failure*[18]. Los métodos *setHorizontalFOV* y *setVerticalFOV* nos permiten fijar los valores de los campos de visión horizontal y vertical respectivamente en unidades de grados sexagesimales. Los valores de los dos últimos parámetros del filtro, la distancia al plano más cercano y la distancia al plano más lejano, son marcados por *setNearPlaneDistance* y *setFarPlaneDistance* en metros. Cabe señalar que, en caso de no hacer uso de alguna de estas funciones, los parámetros tienen un valor por defecto que se les asigna cuando se llama al constructor de la clase. Los prototipos de estas funciones pueden verse en el Código 5.1.

Código 5.1 Prototipo de las funciones que establecen los parámetros del filtro frustum y su aplicación.

```
void setCameraPose(const Eigen::Matrix4f &camera_pose);
void setHorizontalFOV(float hfov);
void setVerticalFOV(float vfov);
void setNearPlaneDistance(float np_dist);
void setFarPlaneDistance(float fp_dist);
void filter(open3d::core::Tensor &indices);
```

Con respecto a la aplicación del filtro, ésta se realiza en la función *filter* a la cual se le pasa como parámetro de salida un *tensor* que contendrá los índices de los elementos que se encuentren dentro del campo de visión determinado por los parámetros que hemos especificado anteriormente. En primer lugar, mediante el uso de los parámetros establecidos, se obtienen las ecuaciones de los planos que van a determinar el campo de visión tal y como se realiza en la librería *PCL*, éstos serán los planos lejano, cercano, superior, inferior, derecho e izquierdo. Una vez que tenemos dichas ecuaciones en vectores de 4 elementos de *Eigen* comienza el proceso de adaptar el filtro a la librería *OPEN3D*, por lo que dichos vectores serán transformados en *tensores*. Para ello, actualmente se puede usar fácilmente la función *EigenMatrixToTensor*, sin embargo, la versión que existía cuando empecé este proyecto me arrojaba un *static assertion failure*. El problema era debido a que esa versión usaba por defecto un alineamiento por filas (*RowMajor*) y no admitía un alineamiento por columnas (*ColMajor*) como requería mi caso, por lo que creé la función *mi_EigenMatrixToTensor* aplicando los cambios necesarios. Contacté con los desarrolladores de *OPEN3D* para explicarles el fallo y enseñarles mi solución[19] y tras unos meses actualizaron la función permitiendo su funcionamiento independientemente del alineamiento[20]. Actualicé *mi_EigenMatrixToTensor* con la nueva versión de *EigenMatrixToTensor* para evitar una reinstalación de toda la librería.

Una vez que obtengo los tensores con las ecuaciones de los planos los almaceno en una sola *struct* llamada *tensores* para poder transferirlos todos de una manera más eficiente a la operación de filtrado. Antes de aplicar el filtro hay que identificar el lugar de almacenamiento en el que se encuentra la nube para ejecutar el filtro en CPU o en GPU según corresponda. Para su ejecución se han creado dos funciones privadas de la clase, una para cada lugar de almacenamiento, cuyos prototipos se muestran en el Código 5.2 y a las que se les pasa como argumento de entrada la estructura *tensores* con las ecuaciones de los planos y como argumento de salida el tensor booleano que contendrá los índices de los elementos buscados. Como sus nombres indican, la función *_filter_CPU* está implementada en un archivo *cpp* y ejecutará el filtro en la CPU mientras que *_filter_CUDA* está implementada en un archivo *cu* y lo ejecutará en la GPU.

Código 5.2 Prototipo de las funciones que aplican el filtro frustum en CPU y en GPU.

```
void _filter_CPU(open3d::core::Tensor &indices, tensores planes);
void _filter_CUDA(open3d::core::Tensor &indices, tensores planes);
```

En ambas implementaciones se empieza inicializando un *tensor* local booleano con todos los elementos a false, siendo este tensor el que irá almacenando los elementos que se encuentren dentro del frustum, y homogeneizando las coordenadas de los puntos de la nube de puntos. Tras esto se realiza el producto matricial de los puntos homogeneizados con cada uno de los planos y se almacenan los resultados en otra estructura *tensores*. El siguiente paso sería la comprobación de que los puntos pertenezcan o no al campo de visión determinado por los planos y para una mayor eficiencia ésto se realiza mediante bucles For paralelizados. Estos bucles no admiten variables ni funciones personalizadas o de ámbito privado, es decir, solo podrán usarse enteros, flotantes, dobles, punteros... Es por eso que es preciso transformar los tensores que contienen los resultados de los productos matriciales y el de los indices deseados a punteros, lo cual puede conseguirse con el método *GetDataPtr<T>* de la clase *Tensor*, siendo *T* el tipo de los datos de los tensores.

Para crear el bucle paralelizado se hace uso de la función *ParallelFor* de la clase *core*, a la que hay que pasarle como parámetros de entrada el lugar de almacenamiento en el que se quiere ejecutar la paralelización, el número de elementos a procesar y una función *lambda* conteniendo la algoritmia que se quiere repetir en el bucle. Para nuestro filtro, la lambda se encargará de realizar la comprobación anteriormente mencionada y en el caso de que el punto esté dentro del frustum pondrá a *true* el elemento correspondiente

dentro del tensor de indices local. Finalmente, se copia este tensor local al tensor de salida *indices*, obteniendo así todos los puntos de la nube de entrada que se encuentren en el frustum. La principal diferencia entre las dos implementaciones es que, debido a que CUDA no permite el uso de lambdas extendidas o *extended lambdas* en el ámbito o *scope* de clases privadas, es necesario ejecutar el bucle paralelizado en una función pública o global no perteneciente a la clase *FrustumCulling* para su implementación en GPU. Para una mejor comprensión de todo lo explicado se adjunta el diagrama de bloques de esta funcionalidad en la Figura 5.1.

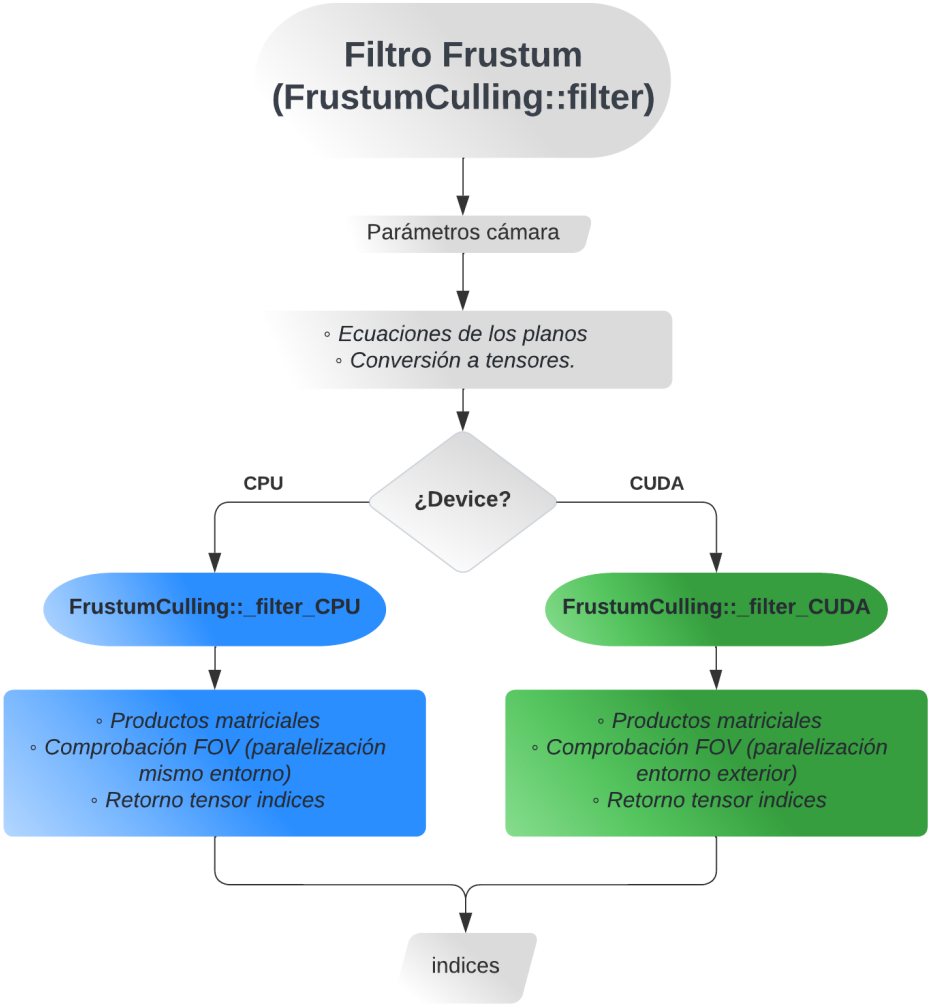

Figura 5.1 Diagrama de flujo del filtro frustum.

En el Código 5.3 se muestra un ejemplo usando este filtro. A partir de una nube de puntos y un lugar de almacenamiento específico (CUDA:0 para GPU o CPU:0 para CPU), transformamos la nube a su equivalente con tensores en el lugar de almacenamiento correspondiente e inicializamos el tensor booleano en el que se guardarán los índices procedentes del filtro frustum. Creamos el objeto de la clase *FrustumCulling* con la nube como parámetro de su constructor, establecemos los parámetros del filtro con sus funciones correspondientes y finalmente aplicamos el filtro. De esta forma, el tensor *selected_indices* nos indicará, con valores a *true*, los elementos de la nube de puntos de entrada que se encuentran dentro del campo de visión elegido.

Código 5.3 Ejemplo de uso del filtro frustum.

```
open3d::geometry::PointCloud original_cloud = ...;
open3d::core::Device device = open3d::core::Device(CUDA:0);
open3d::t::geometry::PointCloud cloud = open3d::t::geometry::PointCloud::
    FromLegacy(original_cloud,open3d::core::Float32,device);
int64_t idx_length = original_device_cloud.GetPointPositions().GetLength();
open3d::core::SizeVector idx_shape = {idx_length};
open3d::core::Tensor selected_indices = open3d::core::Tensor(idx_shape,open3d::
    core::Bool,device);
My_O3D::FrustumCulling frustum = My_O3D::FrustumCulling(cloud);
frustum.setHorizontalFOV(90);
frustum.setVerticalFOV(45);
frustum.setNearPlaneDistance(0.1);
frustum.setFarPlaneDistance(150);
std::vector<float> camera_pose = {1,0,0,0, 0,0,1,0, 0,-1,0,0, 0,0,0,1};
Eigen::Matrix<float, 4, 4, Eigen::RowMajor> cam_pose (camera_pose.data());
frustum.setCameraPose(cam_pose);
frustum.filter(selected_indices);
```

5.2.2 Extractor de índices

Para esta funcionalidad y para el extractor de outliers se ha creado una clase llamada *PointCloud* en cuyo constructor, al igual que en el filtro frustum, se le pasa la nube de puntos que queremos analizar y la almacena en una variable privada.

El extractor de índices, tal y cómo puede verse en el Código 5.4, recibe como parámetros de entrada un tensor booleano y una variable booleana. El tensor tendrá el mismo tamaño que la nube de puntos de entrada y cada uno de sus elementos indicará si el elemento correspondiente de la nube de puntos debe ser mantenido (valor a *true*) o eliminado (valor a *false*). La variable booleana indicará si se quiere o no invertir el valor de los elementos del tensor (de *true* a *false* y viceversa).

Código 5.4 Prototipo de la función extractor de índices con tensores.

```
open3d::t::geometry::PointCloud SelectByIndex (const open3d::core::Tensor &
    indices, bool invert /* = false */);
```

Su funcionamiento se basaría en comprobar el valor de la variable *invert* e invertir o no los valores de los elementos del tensor, almacenando el resultado en un nuevo tensor booleano local. Después se crea una nube de puntos basada en tensores almacenándola en el mismo dispositivo en el que se encuentre la nube de entrada (ya sea CPU o GPU), y en la que se clonarán todos los elementos de la nube entrada cuyo correspondiente valor en el tensor de índices local sea *true*. Por ejemplo, si $indices = [0,1,0,0,1]$ la nube de salida estará compuesta por 2 elementos, el segundo y el quinto elemento de la nube de entrada.

Para acceder a estos elementos se ha hecho uso de una función de alto nivel de la clase *Tensor* llamada *IndexGet()* la cual realiza un indexado avanzado para extraer los elementos dados por un tensor de índices booleano. El funcionamiento descrito se ilustra mediante el diagrama de flujo de la Figura 5.2.

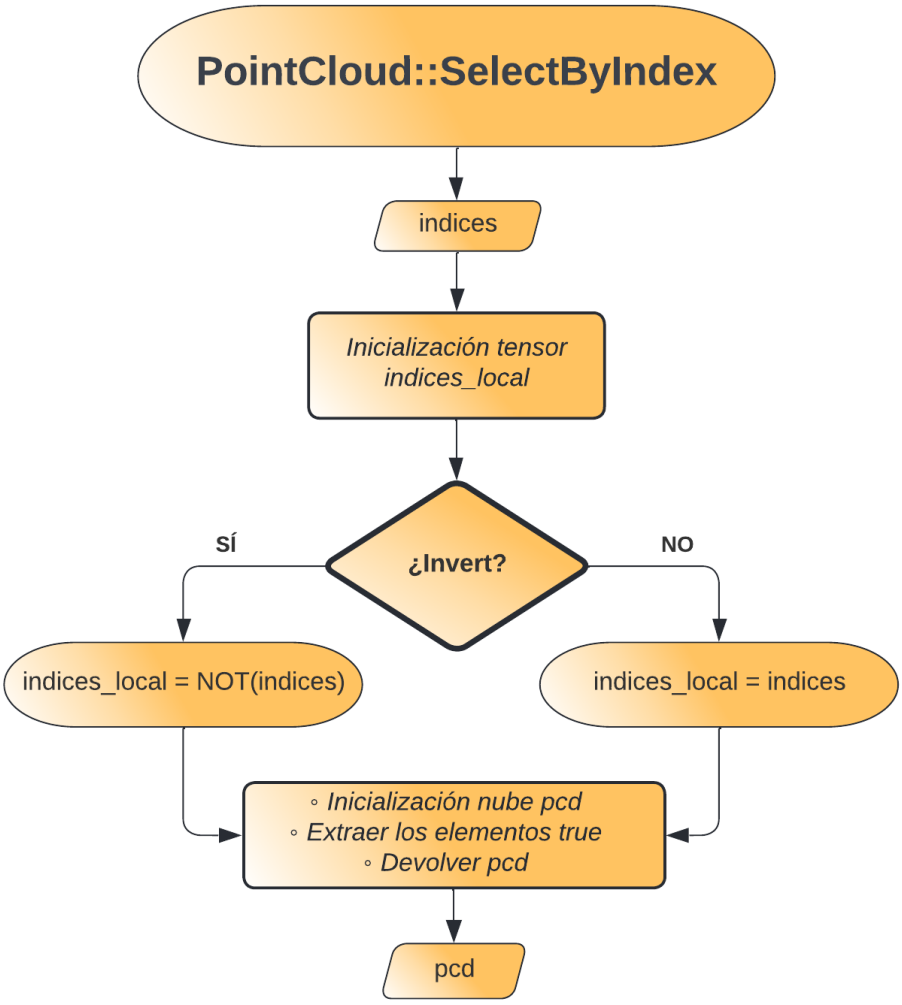

Figura 5.2 Diagrama de flujo del extractor de índices.

Un ejemplo de uso de esta funcionalidad puede verse en el Código 5.5, en el que se parte de una nube de puntos *cloud* y un tensor booleano *selected_indices* con los elementos deseados de dicha nube de puntos. Para filtrar la nube de puntos basta con crear un objeto de la clase *PointCloud* pasándole la nube de puntos a su constructor y realizar una llamada a la función *SelectByIndex* con el tensor de índices como argumento de entrada y la variable de inversión de índices puesta a false. De esta forma, la nueva nube de puntos *filtered_cloud* estaría compuesta únicamente por los elementos dados por el tensor de índices.

Código 5.5 Ejemplo de uso del extractor de índices.

```
open3d::t::geometry::PointCloud cloud = ...;
open3d::core::Tensor selected_indices = ...;
My_O3D::PointCloud target_device = My_O3D::PointCloud(cloud);
open3d::t::geometry::PointCloud filtered_cloud = target_device.SelectByIndex(
    selected_indices,false);
```

5.2.3 Extractor de outliers

La funcionalidad encargada de eliminar los outliers de la nube de puntos tiene el prototipo de función que se muestra en el Código 5.6, en el que se puede observar que dicha función recibirá un entero (*nb_points*),

que representa al número de puntos vecinos que un punto debe de tener como mínimo dentro de un determinado radio para ser considerado un inlier, y como segundo parámetro un flotante de precisión doble (*search_radius*) que representa al radio de búsqueda de vecindad.

Código 5.6 Prototipo de la función extractor de outliers con tensores.

```
std::tuple<open3d::t::geometry::PointCloud, open3d::core::Tensor>
    RemoveRadiusOutliersGPU (int nb_points, double search_radius);
```

Su funcionamiento consiste en primer lugar en almacenar en un *tensor* los puntos de la nube de puntos interna de la clase *PointCloud* que se le ha debido de pasar como parámetro en su constructor, y luego se crea un objeto de la clase *NearestNeighborSearch* a la cual se le ha proporcionado en su constructor el *tensor* de puntos como el dataset de puntos necesario para construir el índice de la búsqueda de vecindad híbrida (*HybridIndex*). A continuación, se realiza una búsqueda de vecinos de tipo híbrida mediante una llamada a la función *HybridSearch*, a la que se le pasan como parámetros el *tensor* de puntos en el que se buscarán los vecinos, el radio de búsqueda *search_radius* y el número de vecinos *nb_points*. Esta búsqueda de vecindad me da como resultado tres *tensores*, uno para los índices de los vecinos dentro de la nube de puntos, otro para las distancias de cada punto a cada uno de sus vecinos y un tercer tensor para indicar el número de vecinos que tiene cada punto. Para la funcionalidad que hemos desarrollado nos sirve únicamente ese tercer tensor, pues será usado para crear un *tensor* booleano y todos aquellos puntos que tengan un número de vecinos mayor o igual que *nb_points* serán marcados a *true* y el resto a *false*. De esta forma, conseguimos un tensor de índices clasificando los elementos de la nube de puntos en *inliers* y *outliers*, el cual servirá para realizar una extracción por índices, mediante la funcionalidad que hemos desarrollado y explicado anteriormente, obteniendo así una nube de puntos compuesta únicamente por los inliers y un *tensor* con los índices de dichos elementos. En la Figura 5.3 se muestra el diagrama de flujo de este algoritmo.

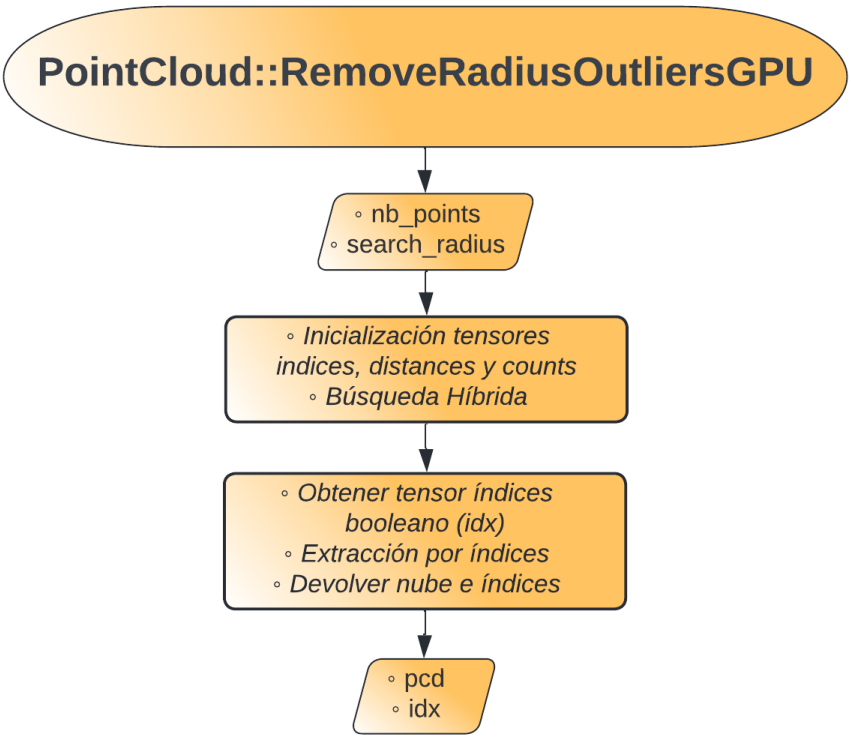

Figura 5.3 Diagrama de flujo del extractor de outliers.

Cabe señalar que la búsqueda de vecinos por radio *FixedRadiusSearch* realiza una búsqueda de todos los vecinos que se encuentran en el radio proporcionado para cada uno de los puntos, mientras que la búsqueda de vecinos híbrida *HybridSearch* realiza una búsqueda de un máximo número de vecinos determinado por su tercer parámetro de entrada (en nuestro caso *nb_points*) para cada uno de los puntos de forma que si, por ejemplo, un punto tuviese 50 vecinos dentro del radio solo buscaría los primeros *nb_points* vecinos. Esto conlleva una gran reducción de carga computacional, lo cual ha sido el principal motivo para escoger este tipo de búsqueda de vecindad frente al otro.

En el Código 5.7 se muestra un ejemplo de uso de esta funcionalidad, en el que se parte de una nube de puntos $filtered\_cloud$, se definen el número máximo de vecinos que se desean buscar (*nb*) y el radio de búsqueda de vecindad (*r*). Se crea un objeto de la clase *PointCloud* pasándole la nube de puntos a su constructor, se inicializan el tensor y la nube de puntos en los que queremos guardar el resultado de aplicar esta funcionalidad y finalmente se realiza una llamada a la función *RemoveRadiusOutliersGPU* con *nb* y *r* como parámetros de entrada. Como resultado, el tensor *index* indicará, con valores a *true*, los elementos de la nube de puntos de entrada considerados inliers y la nube de puntos *inliers_cloud* estará compuesta únicamente por dichos inliers.

Código 5.7 Ejemplo de uso del extractor de outliers.

```
open3d::t::geometry::PointCloud filtered_cloud = ...;
int nb = 3;
double r = 0.5;
My_O3D::PointCloud outliers_gpu = My_O3D::PointCloud(filtered_cloud);
open3d::core::Tensor index;
open3d::t::geometry::PointCloud inliers_cloud;
std::tie(inliers_cloud,index) = outliers_gpu.RemoveRadiusOutliersGPU(nb, r);
```

5.2.4 Extractor de puntos más cercanos

Como se mencionó en la Subsección 5.1.4, para esta funcionalidad se han realizado dos implementaciones de similar diseño, una con tensores y la otra sin ellos, pero ambas con la misma finalidad.

Sin tensores

Para trabajar con nubes de puntos del legado sin tensores se ha desarrollado una única función llamada *GetClosestPoints* encargada de buscar los puntos de una nube entrada más cercanos a una nube de puntos objetivo compuesta por un único punto. Como muestra su prototipo en el Código 5.8, esta función recibe como parámetros de entrada una nube de puntos *source* que contendrá los elementos a analizar, una segunda nube de puntos *target* con el punto que se quiere tomar como referencia para los puntos más cercanos a éste, una variable booleana *neighbourhood* para indicar si se quiere o no buscar los vecinos de cada uno de los puntos más cercanos, un entero *nbs* que representa el número de vecinos máximos a tener en cuenta en la búsqueda de vecindad y una variable flotante de precisión doble *radius* para determinar el radio de la búsqueda de vecindad. Por defecto, la búsqueda de vecindad no estará activada y el número de vecinos y el radio de búsqueda tendrán valores nulos. Como resultado, la función nos devolverá la distancia del punto más cercano al punto de referencia, un vector que contendrá vectores de enteros con los índices del elemento más cercano y sus vecinos (si se indicase), el número de mínimos o puntos más cercanos encontrados y por último el número de vecinos encontrados (0 si no se indicase una búsqueda de vecindad).

Código 5.8 Prototipo del extractor de puntos más cercanos sin tensores.

```
std::tuple<double, std::vector<std::vector<int>>, int, int>
GetClosestPoints (open3d::geometry::PointCloud &source,
                  open3d::geometry::PointCloud &target,
                  bool neighbourhood = false, int nbs = 0, double radius = 0.0);
```

El primer paso que se realiza es calcular las distancias desde cada uno de los puntos de la nube *source* al punto de referencia de la nube *target*, para lo cual se hace uso de la función de alto nivel *ComputePointCloudDistance* que presenta OPEN3D para las nubes de legado. Tras obtener dichas distancias, se busca la

distancia mínima mediante la función de C++ *min_element*, que nos devuelve un iterador hacia el valor de distancia más pequeño del vector, permitiéndonos así obtener la distancia del punto más cercano, que es el primer valor de retorno de esta función.

A continuación, se realiza la búsqueda de los puntos más cercanos dentro del vector de distancias, para ello se comprueba si la distancia correspondiente a cada punto es igual a la mínima obtenida antes, en caso afirmativo se incrementará la variable que indica el número de mínimos encontrados (tercer valor de retorno) y se guardará el índice de dicho elemento en un vector. Como esta búsqueda de mínimos puede suponer una gran carga computacional cuando se trabaja con nubes de puntos de tamaño considerable, ésta se implementó dentro de un bucle for paralelizado mediante directivas de OpenMP y operaciones atómicas para evitar condiciones de carrera. Al haber usado una paralelización de hilos, el vector en el que se almacenaron los elementos mínimos puede estar desordenado por lo que es necesario ordenar dicho vector habiendo eliminado previamente los elementos inicializados que sobran, esto puede conseguirse con las funciones de C++ *erase* y *sort*.

El último paso consiste en obtener el vector de vectores con los índices de los puntos más cercanos, su procedimiento dependerá de si se quiere o no realizar una búsqueda de vecinos. En caso negativo, el procedimiento sería muy sencillo pues bastaría únicamente con recorrer el vector de índices obtenido y guardar sus elementos en vectores por separado dentro de un vector de vectores, que será el segundo valor de retorno de la función. Por otro lado, si se quisiese realizar una búsqueda de vecindad, para cada elemento del vector de índices que obtuvimos en la búsqueda de mínimos realizamos una búsqueda híbrida de vecindad con la función *SearchHybrid* de la clase *kdtree* usando los parámetros *nbs* y *radius*. Los índices que se obtienen de esta búsqueda híbrida de cada punto los almacenamos en el ya mencionado vector de vectores a no ser que se repita el mismo vector de índices (caso posible si hay dos mínimos en las mismas coordenadas) y el número de vecinos se incrementa sin contar el propio mínimo analizado ni los vecinos obtenidos con un vector de índices repetido. Tras esto, ya habríamos obtenido los cuatro parámetros de salida de la función. El diagrama de flujo de esta implementación se muestra en la Figura 5.4.

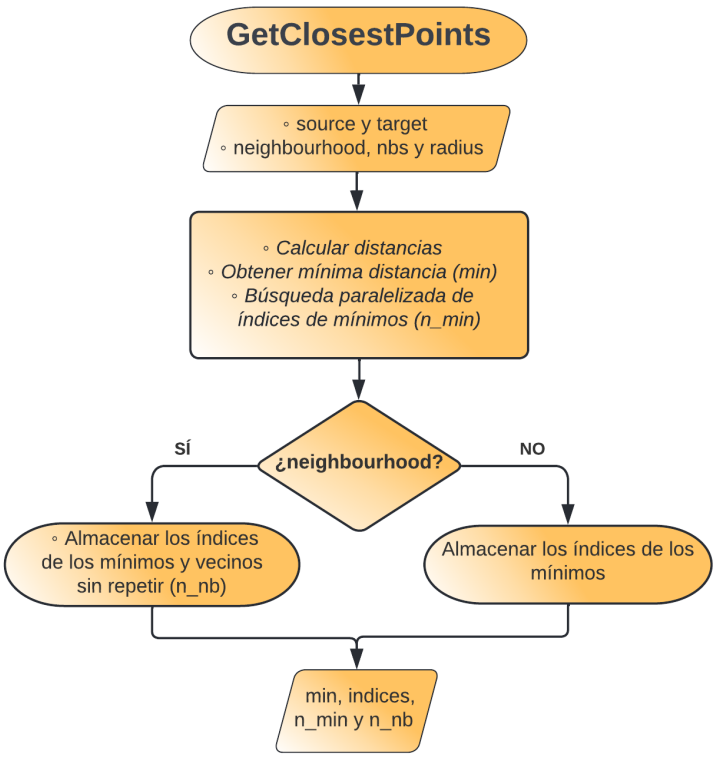

Figura 5.4 Diagrama de flujo del extractor de puntos más cercanos sin tensores.

En el Código 5.9 se proporciona un ejemplo de uso para esta funcionalidad. Se crean la nube *origin* con el punto de referencia con el que se quieren calcular las distancias y la nube *test_c* con una serie de puntos preparada para que haya varios mínimos en diferentes coordenadas y con algunos vecinos alrededor. Si se hace la llamada a la función sin indicar vecindad se obtienen una distancia mínima de 5.38m 4 mínimos en los elementos 1, 2, 6 y 8 de la nube *test_c*. Si la llamada se hace indicando vecindad con un radio de búsqueda de 0.2m y 3 vecinos como máximo, se obtienen mínimos en los elementos 1, 2, 6 y 8 al igual que antes pero con vecinos en los elementos 2, 5, 7 y 9. Cabe señalar que cuando se analiza el elemento 2 no se almacena su vector de índices ni se contabilizan sus vecinos porque se trata del mismo punto que el elemento 1.

Código 5.9 Ejemplo de uso del extractor de puntos más cercanos sin tensores.

```
/* Nubes source y target */
std::vector<Eigen::Vector3d> origin_point {Eigen::Vector3d(0,0,0)};
open3d::geometry::PointCloud origin = open3d::geometry::PointCloud(origin_point
    );
std::vector<Eigen::Vector3d> test ({{5.0, 9.0, 10.0}, {2.0, 3.0, 4.0}, {2.0,
    3.0, 4.0}, {20.0, 8.0, 7.0}, {20.0, 8.0, 7.0}, {3.8, 3.0, 4.0}, {-2.0,
    -3.0, -4.0}, {-3.8, -3.0, -4.0}, {-3.8, 3.0, 4.0}, {-2.0, 3.0, 4.0}, {-3.8,
     3.0, 4.0}});
open3d::geometry::PointCloud test_c = open3d::geometry::PointCloud(test);

/* Inicialización variables de salida y parámetros de entrada para vecindad */
double min_dist;
std::vector<std::vector<int>> indices;
int mins, nbs;
int max_nbs = 3;
double radius = 0.2;

/* --- Sin vecinos --- */
std::tie(min_dist,indices,mins,nbs) = GetClosestPoints(test_c,origin);
/* resultados: min_dist = 5.38; indices=[[1],[2],[6],[8]]; mins=4; nbs=0;*/

/* --- Con vecinos --- */
std::tie(min_dist,indices,mins,nbs) = GetClosestPoints(test_c,origin,true,3,
    _radius);
/* resultados: min_dist = 5.38; indices=[[1, 2, 5],[6, 7],[8, 9]]; mins=4
    (1,2,6,8); nbs=4 (2,5,7,9);*/
```

Con tensores

Para trabajar con tensores y sus nubes de puntos se ha desarrollado esta segunda implementación, la cual tendrá a su misma vez otras dos implementaciones, una para ejecutar en CPU y otra para ejecutar en GPU. Se ha creado una clase llamada *ClosestPoints_t* en la que se definirán todas las funciones necesarias y en cuyo constructor se le indica la nube de puntos a analizar y que será la nube *source* para todas las funciones de la clase. Cabe señalar que el procedimiento de esta implementación es prácticamente idéntico al de la implementación sin tensores.

En la anterior implementación podíamos hacer uso de la función de OPEN3D *ComputePointCloudDistance* para calcular las distancias entre dos nubes de puntos, sin embargo, para tensores no había ninguna función equivalente por lo que fue necesario crear nuestra propia función para poder usarla luego en la función principal. Esta función ha sido llamada *ComputePointCloudDistance_Tensor* y lo primero que hace es comprobar el lugar de almacenamiento en el que se encuentra la nube de puntos interna de la clase. Si la nube está en CPU se seguirá en dicha función pues está implementada en un archivo *cpp*, pero si la nube estuviese en GPU se hace una llamada a *_ComputePointCloudDistance_CUDA*, que es una función privada de la clase, para ejecutar el código en un archivo *cu*. Los prototipos de ambas funciones se muestran en el Código 5.10. El funcionamiento de ambos casos es el mismo y se basa en realizar una búsqueda de un único

vecino dentro de la nube *source* para el punto de referencia de la nube *target* mediante el uso de la función *KnnSearch* de la clase *NearestNeighborSearch*. Ésta búsqueda nos permite obtener un tensor con las distancias de cada uno de los puntos de la nube *source* con respecto al punto de referencia de la nube *target*. Su funcionamiento es ilustrado en la Figura 5.5.

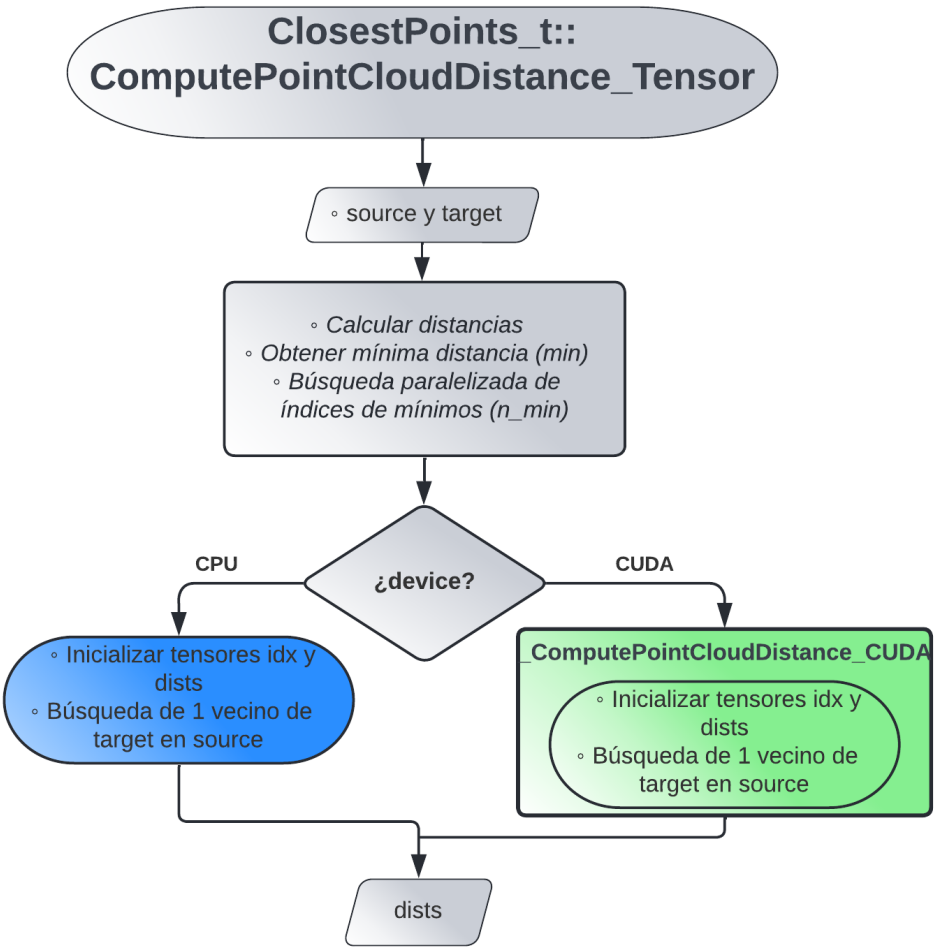

Figura 5.5 Diagrama de flujo del cálculo de la distancia entre dos nubes de puntos basadas en tensores.

Código 5.10 Prototipos de funciones para el cálculo de distancias entre nubes de puntos con tensores.

```
open3d::core::Tensor ComputePointCloudDistance_Tensor (open3d::t::geometry::
    PointCloud &target);
open3d::core::Tensor _ComputePointCloudDistance_CUDA (open3d::t::geometry::
    PointCloud &target);
```

Tras haber explicado esta función ya podemos analizar la función principal *GetClosestPoints_Tensor* encargada de obtener los puntos más cercanos. Al igual que la anterior función, ésta comienza comprobando el lugar de almacenamiento de la nube interna de la clase. De la misma forma, si la nube está en CPU se seguirá en dicha función pues está implementada en un archivo *cpp*, pero si la nube estuviese en GPU se hace una llamada a *_GetClosestPoints_CUDA*, que es una función privada de la clase, para ejecutar el código en un archivo *cu*. Sus prototipos pueden verse en el Código 5.11, reciben como parámetros la nube *target* con el punto de referencia, una variable booleana *neighbourhood* para indicar si se quiere o no analizar la vecindad de los mínimos, un entero *nbs* indicando el número de vecinos como máximo para la búsqueda de vecinos y una variable flotante de precisión doble *radius* para el radio de búsqueda. Por defecto, la búsqueda

de vecindad no estará activada y el número de vecinos y el radio de búsqueda tendrán valores nulos. Como retorno, estas funciones devuelven la distancia al punto más cercano, un tensor con los índices de los puntos más cercanos y sus vecinos (si se indicase), el número de mínimos encontrados y el número de vecinos detectados (0 si no se indicase una búsqueda de vecindad).

Código 5.11 Prototipos de funciones para la obtención de los puntos más cercanos con tensores.

```
std::tuple<double, open3d::core::Tensor, int, int>
       GetClosestPoints_Tensor (open3d::t::geometry::PointCloud &target,
                                bool neighbourhood = false, int nbs = 0, double
                                    radius = 0.0);

std::tuple<double, open3d::core::Tensor, int, int>
       _GetClosestPoints_CUDA (open3d::t::geometry::PointCloud &target,
                               bool neighbourhood, int nbs, double radius);
```

El funcionamiento de ambos casos es idéntico pero con una pequeña diferencia. Ambas funciones empiezan obteniendo el tensor con las distancias de cada punto de la nube *source* al punto de referencia de la nube *target* mediante llamadas a las funciones *ComputePointCloudDistance_Tensor* para el caso de CPU y *_ComputePointCloudDistance_CUDA* para GPU. Con el tensor de distancias se puede calcular el valor mínimo de éste mediante el método *Min* de la clase *Tensor* obteniendo así la distancia al punto más cercano, que será el primer valor de retorno de la función. Es ahora cuando surge la diferencia entre las dos implementaciones, que es originada por el mismo motivo que se explicó en la Subsección 5.2.1 con el filtro Frustum, relacionada con el distinto funcionamiento de las *extended lambdas* en los entornos de los bucles *ParallelFor* con CPU y GPU. Para obtener un tensor que contenga los índices de los mínimos se hace necesario el uso de bucles paralelizados, pero antes hace falta crear un tensor entero *indices_int* y otro booleano *indices_bool* y obtener los punteros a estos dos tensores y al de distancias calculado anteriormente. Para el bucle, se crea una lambda en la que se busca en el tensor de distancias aquellos elementos que sean iguales que el mínimo calculado y en caso afirmativo se pone a *true* dicho elemento en el tensor booleano y se guarda su índice en el tensor entero. En el caso de la CPU este bucle se ejecuta en la misma función principal pero en el caso de CUDA éste se realiza en la función *MinimumsSearch* externa a la clase *ClosestPoints_t*. En otro tensor almacenamos los índices enteros guardados en *indices_int* que estén marcados a *true* en *indices_bool*, y lo pasamos a un vector estándar para un posterior acceso más sencillo.

El último paso es comprobar si se ha indicado la decisión de realizar una búsqueda de vecindad o no. En caso negativo, se almacenará en el tensor de salida los índices obtenidos antes y se calculará el número de mínimos presentes para el tercer valor de retorno. En el caso de que sí sea necesaria hacer una búsqueda de vecindad, para cada uno de los puntos de la nube determinados por los índices obtenidos, se realiza una *HybridSearch* en la nube interna *source*, obteniendo así un tensor con el índice del mínimo analizado y el de sus vecinos. Tras ello se comprueba si dicho tensor estuviese ya repetido por la presencia de otro mínimo en las mismas coordenadas, en cuyo caso contrario se almacenaría el tensor obtenido en el tensor de salida y se aumentaría el número de vecinos. Con esto se habrían obtenido los cuatro parámetros de salida necesarios y se procederá a agruparlos para devolver el conjunto como salida de la función. El funcionamiento de esta implementación con tensores se ilustra mediante un diagrama de flujos en la Figura 5.6.

En el Código 5.12 puede observarse cómo se usaría esta funcionalidad. Se crean la nube *origin* con el punto de referencia con el que se quieren calcular las distancias y la nube *test_c* con una serie de puntos preparada para que haya varios mínimos en diferentes coordenadas y con algunos vecinos alrededor. Ambas nubes de puntos se almacenan en el lugar especificado por *device*, pudiendo tratarse de CUDA o CPU. Se inicializan las variables de salida, los parámetros de entrada para la vecindad y se crea un objeto de la clase *ClosestPoints_t* pasándole la nube *test_c* para que todas las funciones de la clase puedan acceder a dicha nube. Finalmente, se realiza la llamada a la función pudiendo elegir entre realizar búsqueda de vecinos o no. Los resultados obtenidos en esta implementación son, matemáticamente, los mismos que en la implementación sin tensores, solo cambia la estructura o tipo de dato en el que se obtienen los índices de los puntos más cercanos.

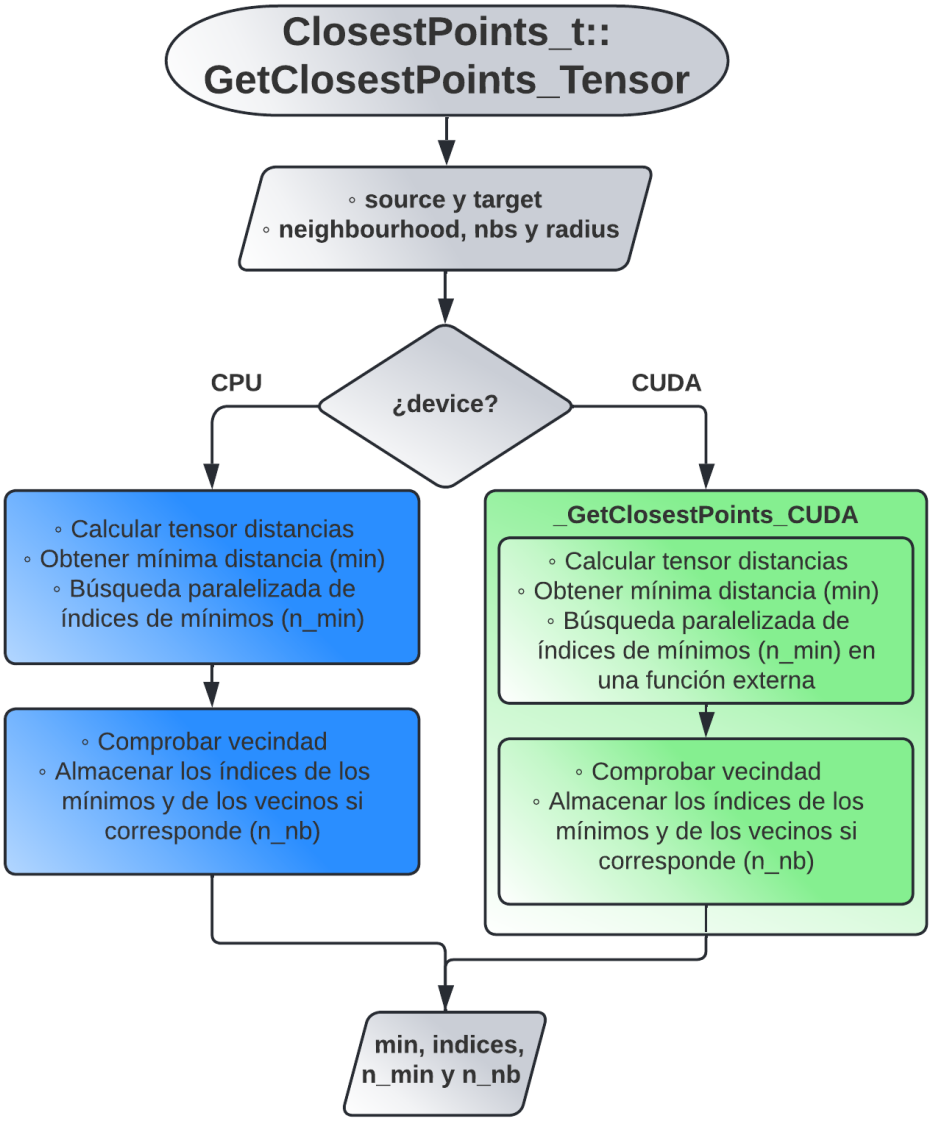

Figura 5.6 Diagrama de flujo del extractor de puntos más cercanos con tensores.

Código 5.12 Ejemplo de uso del extractor de puntos más cercanos con tensores.

```
/* Nubes source y target */
open3d::core::Device device = open3d::core::Device("CUDA:0"); // o "CPU:0"
open3d::core::Tensor ref = open3d::core::Tensor::Init<float>({{0.0, 0.0, 0.0}},
    device);
open3d::t::geometry::PointCloud origin = open3d::t::geometry::PointCloud(ref);
open3d::core::Tensor test = open3d::core::Tensor::Init<float>({{5.0, 9.0,
    10.0}, {2.0, 3.0, 4.0}, {2.0, 3.0, 4.0}, {20.0, 8.0, 7.0}, {20.0, 8.0,
    7.0}, {3.8, 3.0, 4.0}, {-2.0, -3.0, -4.0}, {-3.8, -3.0, -4.0}, {-3.8, 3.0,
    4.0}, {-2.0, 3.0, 4.0}, {-3.8, 3.0, 4.0}}, device);
open3d::t::geometry::PointCloud test_c = open3d::t::geometry::PointCloud(test);

/* Inicialización variables de salida y parámetros de entrada para vecindad */
double min_dist;
open3d::core::Tensor indices;
int mins, nbs;
```

```
int max_nbs = 3;
double radius = 0.2;

/* Constructor de la clase */
ClosestPoints_t CPoints = ClosestPoints_t(test_c);

/* --- Sin vecinos --- */
std::tie(min_dist,indices,mins,nbs) = CPoints.GetClosestPoints_Tensor(origin);
/* resultados: min_dist = 5.38; indices=[[1],[2],[6],[8]]; mins=4; nbs=0;*/

/* --- Con vecinos --- */
std::tie(min_dist,indices,mins,nbs) = CPoints.GetClosestPoints_Tensor(origin,
    true, 3, _radius);
/* resultados: min_dist = 5.38; indices=[[1, 2, 5],[6, 7],[8, 9]]; mins=4
    (1,2,6,8); nbs=4 (2,5,7,9);*/
```

5.2.5 Creador de nube de puntos aleatoria

Para esta funcionalidad también se han desarrollado dos implementaciones tal y como se comentó en la Subsección 5.1.5, una para trabajar sin tensores y otra con ellos.

Sin tensores

Para crear una nube de puntos del legado aleatoria se ha creado la función *makeRandCloud_Leg*, a la cual se le pasarán como parámetros el número de puntos deseados *n_points* y dos vectores double de 3 dimensiones de Eigen con los límites para cada las coordenadas X, Y y Z (uno para los valores máximos y otro para los mínimos) y devolverá una nube de puntos del legado con puntos aleatorios. Esta función se apoyará en otra auxiliar llamada *VectorRand* que será la encargada de crear un vector con todos los puntos aleatorios a partir de un tamaño específico, un vector de generaciones de números aleatorios y dos vectores para los límites máximos y mínimos. Los prototipos de estas funciones pueden verse en el Código 5.13.

Código 5.13 Prototipos de funciones para la creación de nubes de puntos aleatorios sin tensores.

```
open3d::geometry::PointCloud makeRandCloud_Leg(int64_t n_points, Eigen::
    Vector3d max, Eigen::Vector3d min);

std::vector<Eigen::Vector3d> VectorRand(int size, std::vector<std::mt19937>
    gens, const Eigen::Vector3d& low_range, const Eigen::Vector3d& high_range);
```

La función *makeRandCloud_Leg* comenzará creando una semilla aleatoria e inicializando tres generadores de números pseudoaleatorios para las coordenadas XYZ mediante las funciones de C++ *random_device* y *mt19937*. El siguiente paso que realiza es agrupar los tres en un vector *gens* y, junto a *n_points*, *min* y *max*, pasarlos como parámetros en la llamada a la función *VectorRand*, la cual nos devolverá un vector con los puntos aleatorios ya generados y con el que podremos crear la nube aleatoria de salida. Para crear dicho vector de puntos aleatorios, *VectorRand* obtendrá un número aleatorio distinto basado en la secuencia del generador correspondiente para cada una de las coordenadas XYZ de los *size* puntos, haciendo uso de la función de C++ *uniform_real_distribution*. Para reducir la carga computacional de este proceso, éste se ha implementado en una *lambda* para poder ejecutarla en un bucle *ParallelFor* en la CPU. El diagrama de flujo de esta implementación se muestra en la Figura 5.7.

Para usar esta funcionalidad se define un ejemplo en el Código 5.14, en el que se parte de una nube del legado *filtered_cloud* a la que se le calcularán su tamaño y sus límites máximos y mínimos para las coordenadas XYZ. Se creará una nube aleatoria *random_cloud_leg* con un tamaño cuatro veces menor que la anterior nube y definida por los límites calculados anteriormente. El motivo de este algoritmo era poder medir la eficacia del extractor de outliers, y para ello podemos mezclar la nube aleatoria *random_cloud_leg* que hemos obtenido y mezclarla con la *filtered_cloud* para luego aplicar el extractor de outliers a esa nube mezclada y así poder comparar los outliers eliminados con los puntos aleatorios que se le añadieron.

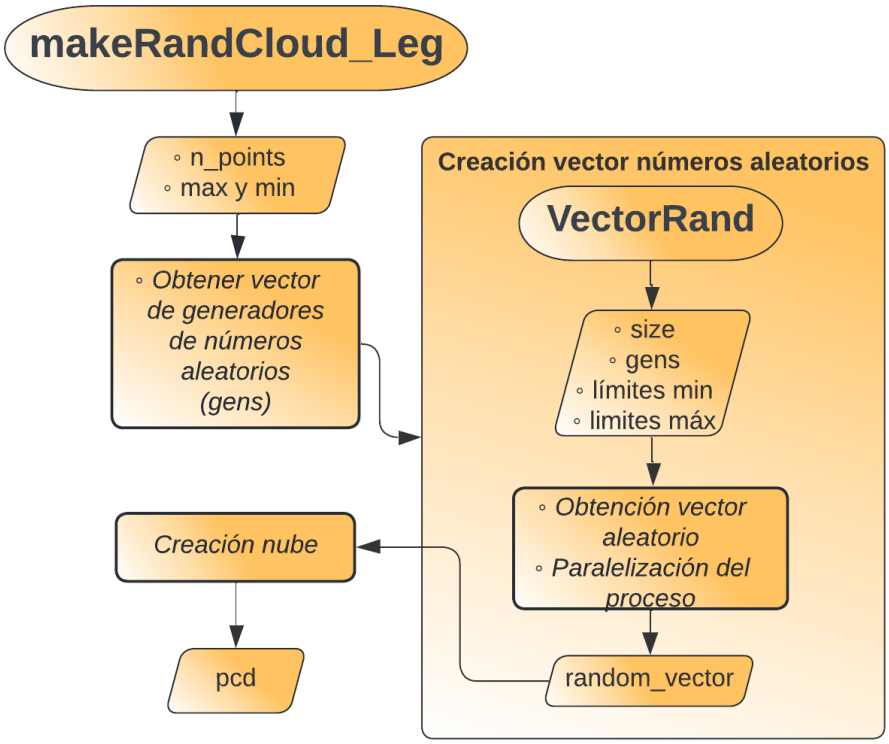

Figura 5.7 Diagrama de flujo del creador de nube de puntos aleatoria sin tensores.

Código 5.14 Ejemplo de uso del creador de nubes de puntos aleatorias sin tensores.

```
/* Inicializa nubes */
open3d::geometry::PointCloud filtered_cloud = ...;
open3d::geometry::PointCloud random_cloud_leg, noisy_cloud_leg;
std::shared_ptr<open3d::geometry::PointCloud> inliers_noisy_cloud;

/* Almacena variables de tamaños y límites */
size_t size_l = filtered_cloud.points_.size();
size_t size_l_random = size_l/4;
Eigen::Vector3d max_values = filtered_cloud.GetMaxBound();
Eigen::Vector3d min_values = filtered_cloud.GetMinBound();

/* Crea la nube de puntos aleatorios */
random_cloud_leg = makeRandCloud_Leg(size_l_random, max_values, min_values);

/* Mezcla la nube aleatoria con la filtrada */
std::vector<Eigen::Vector3d> noisy_vector; noisy_vector.reserve(size_lid+
    size_l_random);
noisy_vector = filtered_cloud->points_;
noisy_vector.insert(noisy_vector.end(), random_cloud.points_.begin(),
    random_cloud.points_.end());
noisy_cloud_leg = open3d::geometry::PointCloud(noisy_vector);

/* Aplica el extractor de outliers */
std::tie(inliers_noisy_cloud,cpu_idx) = noisy_cloud_leg.RemoveRadiusOutliers(
    _neighbours, _radius);
```

Con tensores

Para trabajar con nubes de puntos basadas en tensores se desarrolló la función *makeRandCloud_T* que recibirá como parámetros de entrada un determinado número de puntos (*n_points*) y dos vectores de tres dimensiones de *Eigen* con los límites máximos y mínimos de cada una de las coordenadas XYZ (*max* y *min*). Para obtener los puntos aleatorios esta función se apoyará en otra auxiliar llamada *TensorRand* que recibirá como parámetros de entrada las dimensiones deseadas del tensor (*shape*), una semilla aleatoria (*seed*), un par de valores representando a los límites máximo y mínimo (*range*), el tipo de dato deseado (*dtype*) y el lugar de almacenamiento (*device*). Los prototipos de estas funciones pueden comprobarse en el Código 5.15.

Código 5.15 Prototipos de funciones para la creación de nubes de puntos aleatorios con tensores.

```
open3d::t::geometry::PointCloud makeRandCloud_T(int64_t n_points, open3d::core
    ::Device device, Eigen::Vector3d max, Eigen::Vector3d min);

open3d::core::Tensor TensorRand(const open3d::core::SizeVector& shape,
            size_t seed,
            const std::pair<open3d::core::Scalar,open3d::core::Scalar>& range,
            open3d::core::Dtype dtype, const open3d::core::Device& device);
```

El funcionamiento de este algoritmo consistiría en realizar tres llamadas a *TensorRand*, una para cada coordenada, pasándole una semilla aleatoria distinta a cada una y sus límites correspondientes. El tamaño del tensor vendrá dado por *n_points* filas y 1 columna, su tipo de datos será flotante y el lugar de almacenamiento será el que se especifique en *device*. Esta función auxiliar, basada en un *benchmark*[21] de OPEN3D que no ha llegado a implementarse todavía en la librería, operará inicialmente con un tensor en la CPU, paralelizará el proceso de obtención de números aleatorios con un bucle *ParallelFor* y copiará dicho tensor al lugar de almacenamiento especificado. Tras realizar las tres llamadas a *TensorRand* habremos obtenido un tensor de puntos aleatorios para cada una de las coordenadas XYZ, por lo que el siguiente paso es agruparlos todos en un único tensor por columnas, mediante el método *Append* de la clase *Tensor*, y con este tensor conjunto se crea finalmente la nube de puntos aleatorios basada en tensores. En la Figura 5.8 se muestra el diagrama de flujo de esta implementación.

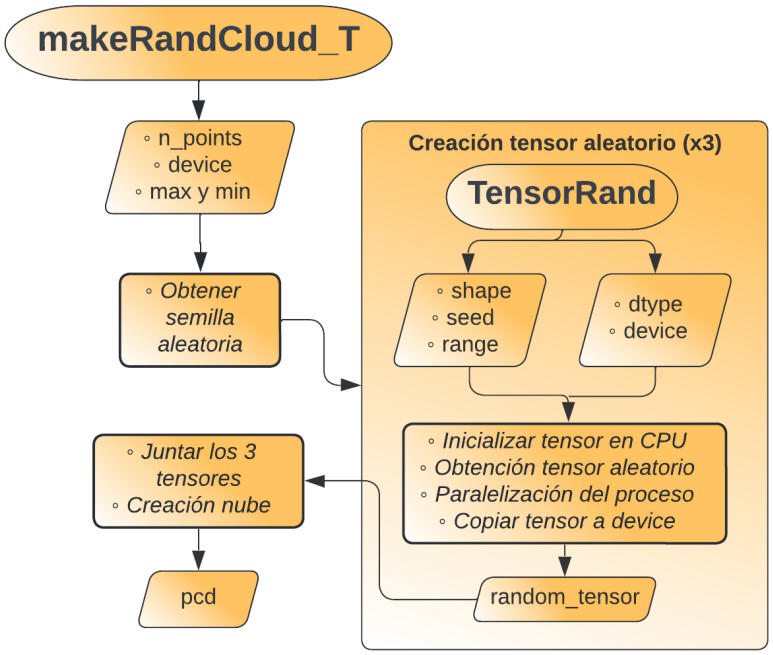

Figura 5.8 Diagrama de flujo del creador de nube de puntos aleatoria con tensores.

Para demostrar cómo se usa esta funcionalidad se adjunta el Código 5.16, en el que se parte de la nube *filtered_cloud* a la que le calcularemos sus límites máximos y mínimos para XYZ y su tamaño. Cabe señalar que estos límites se obtienen en tensores unidimensionales de 3 elementos y hacen falta redimensionarlos de forma que tengan 1 fila y 3 columnas y así poder convertirlos de tensor a vector de Eigen. Se creará la nube aleatoria *random_cloud_t* con un tamaño una cuarta parte menor que *filtered_cloud*, almacenada en la GPU y con los límites anteriormente calculados. Al igual que se comentó en la implementación sin tensores, podemos mezclar la nube aleatoria *random_cloud_t* que hemos obtenido y mezclarla con la *filtered_cloud* para luego aplicar el extractor de outliers a esa nube mezclada y así poder comparar los *outliers* eliminados con los puntos aleatorios que se le añadieron.

Código 5.16 Ejemplo de uso del creador de nubes de puntos aleatorias con tensores.

```
/* Inicializa nubes */
open3d::t::geometry::PointCloud filtered_cloud = ...;
open3d::t::geometry::PointCloud random_cloud_t, noisy_cloud_t,
    inliers_cloud_noise;
open3d::core::Tensor index_noise;
open3d::core::Device device = open3d::core::Device("CUDA:0");

/* Calcula valores max y min de la nube filtrada, las dimensiones del tensor
    deben ser compatibles con {None,3} por lo que es necesario redimensionarlos
     de {3} a {1,3} */
open3d::core::Tensor max_values_t= filtered_cloud.GetMaxBound().Reshape({1,3});
open3d::core::Tensor min_values_t= filtered_cloud.GetMinBound().Reshape({1,3});
Eigen::Vector3d max_values = *open3d::core::eigen_converter::
    TensorToEigenVector3dVector(max_values_t).data();
Eigen::Vector3d min_values = *open3d::core::eigen_converter::
    TensorToEigenVector3dVector(min_values_t).data();
int64_t size = filtered_cloud.GetPointPositions().GetLength();
int64_t size_random = size/4;

/* Crea la nube de puntos aleatorios */
random_cloud_t = makeRandCloud_T(size_random, device, max_values, min_values);

/* Mezcla la nube aleatoria con la filtrada */
noisy_cloud_t = filtered_cloud.Append(random_cloud_t);

/* Aplica el extractor de outliers */
My_O3D::PointCloud outliers_gpu_noise = My_O3D::PointCloud(noisy_cloud_t);
std::tie(inliers_cloud_noise,index_noise) = outliers_gpu_noise.
    RemoveRadiusOutliersGPU(_neighbours, _radius);
```

5.3 Conclusión

A lo largo de este apartado se han explicado detalladamente las múltiples implementaciones realizadas para las diferentes funcionalidades escogidas y se ha querido hacer notar el trabajo e investigación que han sido necesarios para desarrollar todos estos algoritmos en el entorno de programación de dos lugares de almacenamiento distintos, buscando a la misma vez la máxima eficiencia posible en cada uno de ellos. Un aspecto a recalcar es que las implementaciones con tensores son las más costosas en cuanto a esfuerzo y comprensión, pues es necesario estudiar con detenimiento todas sus propiedades, dimensiones y operaciones que permite esta clase, además de las diferencias a nivel de código que hacen falta tener en cuenta para implementar el algoritmo en la CPU o en la GPU, sobre todo cuando se trata de paralelizar los procesos.

6 Resultados

En este capítulo se van a mostrar y analizar los resultados obtenidos de las ejecuciones de los algoritmos desarrollados tanto en CPU como en GPU. También se incluirán los resultados obtenidos ejecutando la algoritmia en una *NVIDIA Jetson TX2* de la misma forma en la que se obtuvieron el resto.

El conjunto de características de la CPU y la GPU tanto del portátil OMEN como de la Jetson TX2 se muestra en la Tabla 6.1. Donde "arq." hace referencia a la micro-arquitectura que usa la GPU y "cap." a la capacidad de cómputo de CUDA que presenta la GPU.

Tabla 6.1 Conjunto de características de las CPUs y GPUs de los dispositivos usados.

Dispositivo	Características CPU	Características GPU
HP OMEN 15-CE0XX	Intel Core i7-7700HQ (4 núcleos x 2.8GHz, 8 hilos)	NVIDIA GeForce GTX 1050 Mobile [22] (arq. Pascal, cap. 6.1, 1493 MHz, 640 núcleos NVIDIA CUDA)
Kit desarrollador NVIDIA Jetson TX2	Dual-core NVIDIA Denver 2 (2 núcleos x 2GHz, 2 hilos) Quad-core ARM Cortex-A57 (4 núcleos x 2GHz, 4 hilos)	Jetson TX2 GPU [23] (arq. Pascal, cap. 6.2, 1300 MHz, 256 núcleos NVIDIA CUDA)

La Jetson TX2 permite 5 modos o perfiles distintos para ajustar su rendimiento y consumo de energía. Como la finalidad de este proyecto es buscar la máxima eficiencia posible se ha escogido el modo 0 o Max-N, el cual establece la frecuencia máxima de la GPU a 1300 MHz y la mínima a 114 MHz. Para aumentar aún más el rendimiento, se ha ejecutado el script *jetson_clocks* proporcionado por NVIDIA en sus dispositivos para establecer la frecuencia de la GPU mínima al mismo valor que la máxima, en nuestro caso 1300 MHz. Cabe señalar también que las pruebas en el portátil OMEN se hicieron con la versión 11.0 de CUDA pero en la Jetson TX2 hubo que hacerlas en la versión 10.2 ya que tuvo que ser configurada con *Jetpack 4.6* (la versión en desarrollo *Jetpack 5* ya admite CUDA 11.0 pero en el momento de estas pruebas no existía aún).

Para poder realizar comparaciones entre todos los casos posibles se han generado una serie de gráficas para una mejor apreciación visual de las diferencias existentes en los resultados. Para generar estas gráficas se ha usado el módulo de *Python matplotlib*, usando como datos los obtenidos de los archivos *csv* que han sido generados tras ejecutar el nodo de ROS que hace uso de los algoritmos desarrollados, teniendo en cuenta todas las variantes posibles. Para poder obtener esas métricas se definieron un par de funciones en el nodo de ROS para almacenar y procesar las estadísticas de cada uno de los algoritmos ejecutados. Las métricas a analizar son los tiempos de ejecución, los tamaños de las nubes de puntos procesadas, el número de iteraciones ejecutadas y el porcentaje de outliers eliminados.

6.1 Nodo de ROS

El nodo de ROS que se ha mencionado está implementado de forma que en un primer lugar, después de recibir una nube de puntos por el topic al que está suscrito, se ejecuta un filtro *frustum* sobre dicha nube para obtener los índices de los puntos que se encuentren dentro del campo de visión especificado. Para los resultados obtenidos se ha establecido unos ángulos de visión vertical y horizontal de 179º, el plano cercano a 0.1 m, el plano lejano a 300 m y la orientación de la cámara siguiendo el marco de referencia del *LIDAR*, que se trata de un sistema de coordenadas *FRD* (X hacia delante, Y derecha, Z abajo). Desde el archivo launch puede escogerse si aplicar un segundo filtro frustum para aumentar el campo de visión con los mismos parámetros y obtener así una vista completa, dicha opción será usada en los escenarios con el dron en el suelo y en el de la nube pequeña por el contraste de resultados.

El siguiente paso del nodo es comprobar dos variables del launch que especificarán el lugar de almacenamiento deseado y si se quiere o no trabajar con tensores, lo cual marcará la ejecución del resto de algoritmos. Si se indica CPU sin tensores se ejecutarán los siguientes algoritmos con las implementaciones que trabajan con nubes del legado mientras que si se indica CPU con tensores o GPU se ejecutarán con las implementaciones que trabajan con nubes de puntos basadas en tensores y en el lugar de almacenamiento correspondiente.

Una vez comprobado lo anterior se ejecuta el *extractor de índices* para obtener la nube filtrada mediante los índices obtenidos del *frustum*. A la nube filtrada se le aplica el *extractor de outliers*, con 2 vecinos y un radio de búsqueda de 0.6 m, para eliminar el posible ruido del *LIDAR* y a ésta se le inserta una nube aleatoria que depende de su tamaño y sus límites. Se vuelve a ejecutar el extractor de outliers, pero esta vez al conjunto de las dos nubes de puntos, y se calcula el porcentaje de puntos aleatorios eliminados. La nube de puntos restante es usada finalmente por el algoritmo del *extractor de puntos más cercanos*, una ejecución sin *vecindad* y otra con ella. Lo último que realiza el nodo es el volcado de todas las métricas obtenidas a un archivo *csv*, además de mostrarlas por la terminal pasados 4 segundos.

6.2 Dron en el suelo

6.2.1 OMEN

Frustum individual

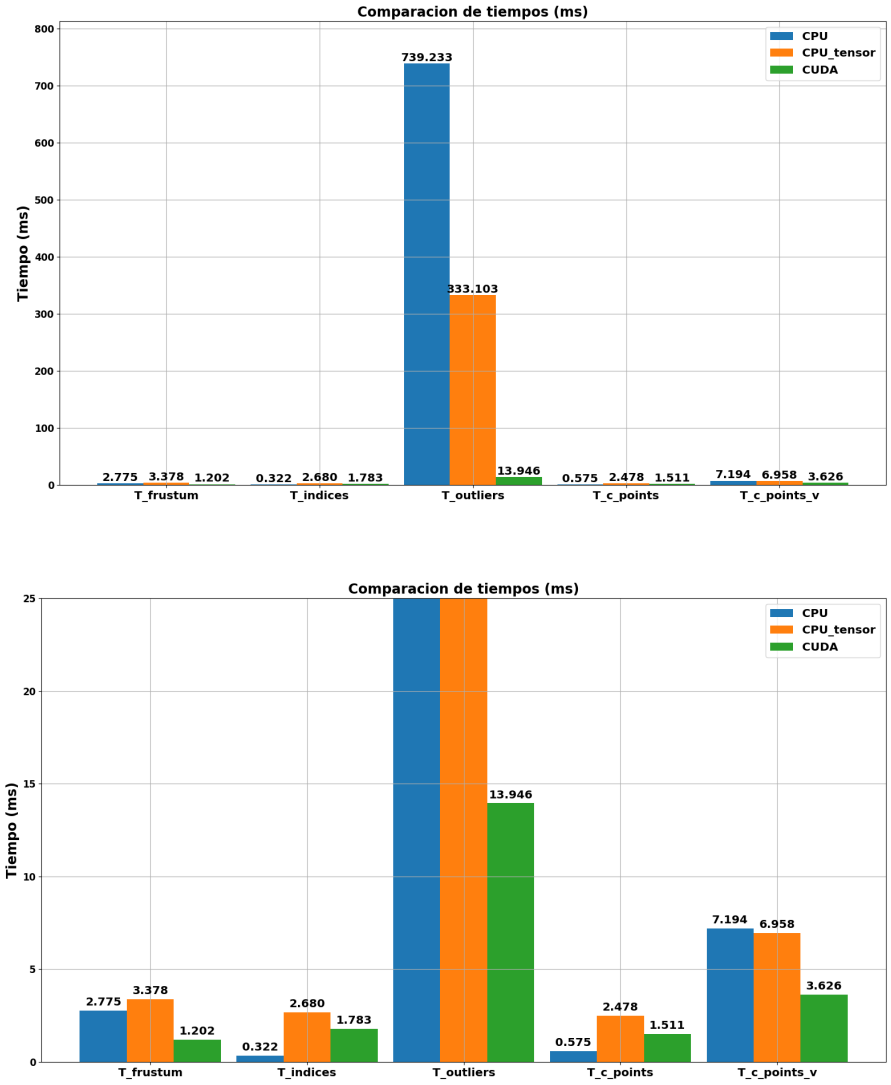

Figura 6.1 Tiempos de ejecución con frustum individual - OMEN - Dron en el suelo.

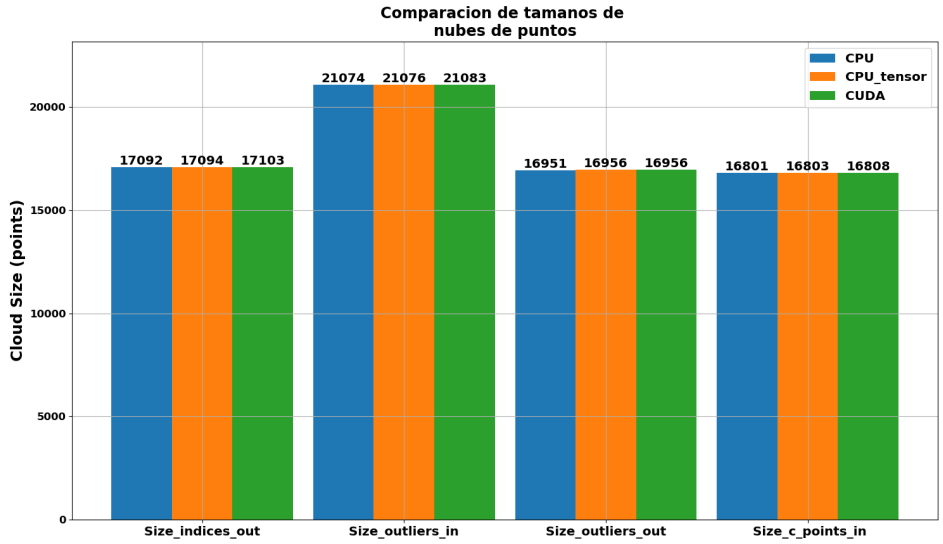

Figura 6.2 Tamaños de nubes con frustum individual - OMEN - Dron en el suelo.

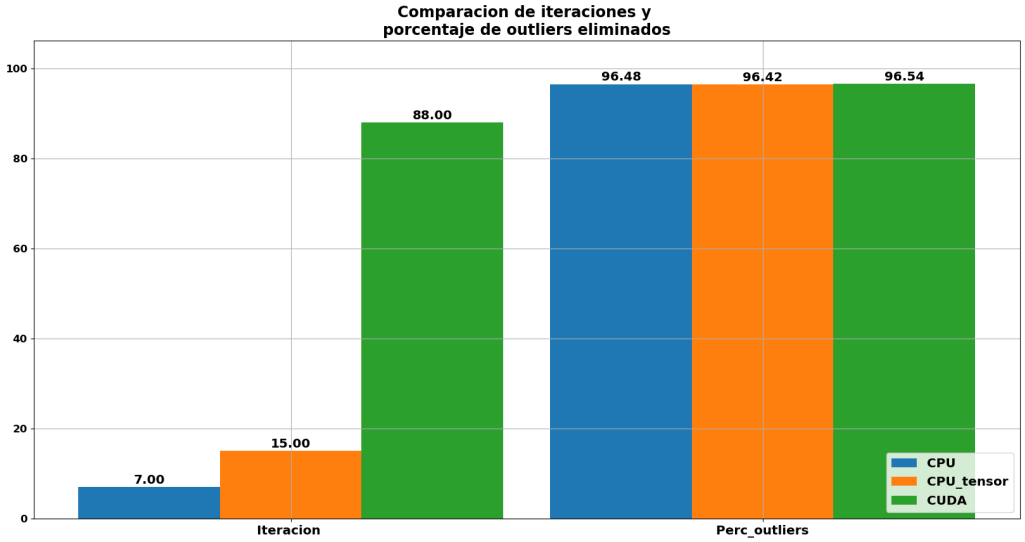

Figura 6.3 Iteraciones y porcentaje outliers con frustum individual - OMEN - Dron en el suelo.

Frustum doble

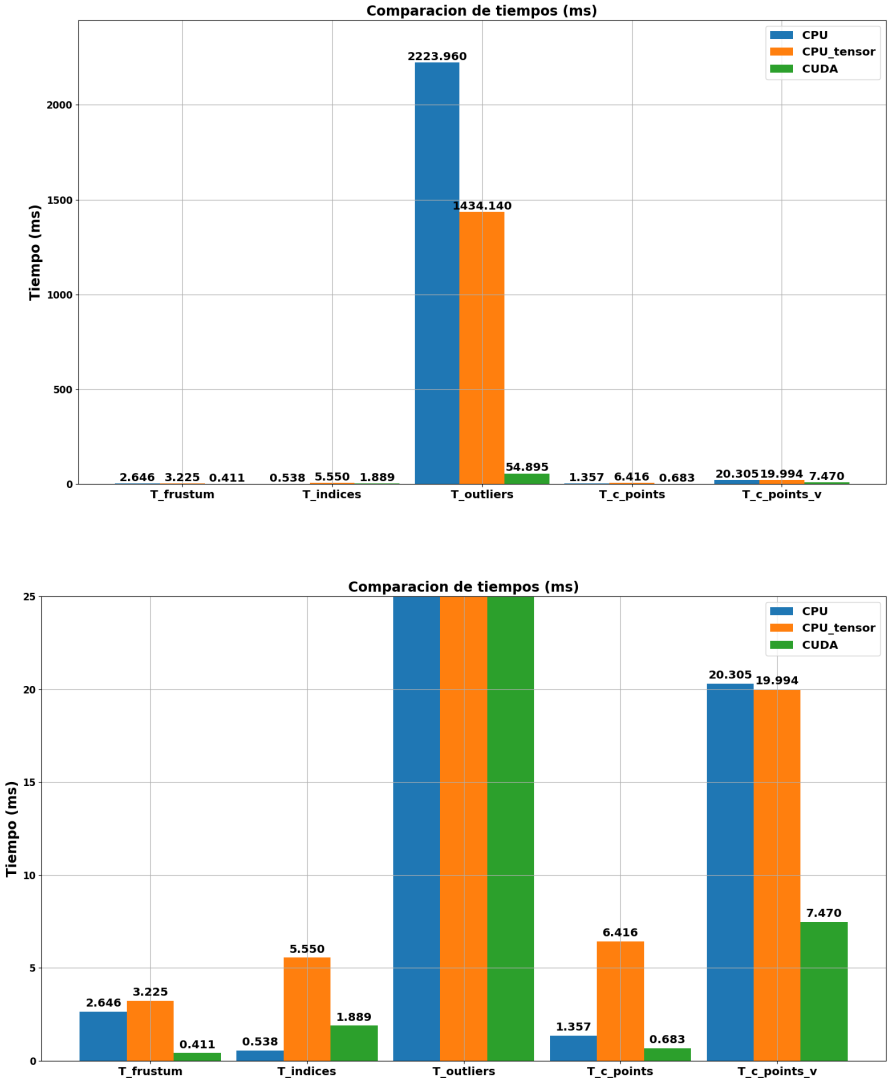

Figura 6.4 Tiempos de ejecución con frustum doble - OMEN - Dron en el suelo.

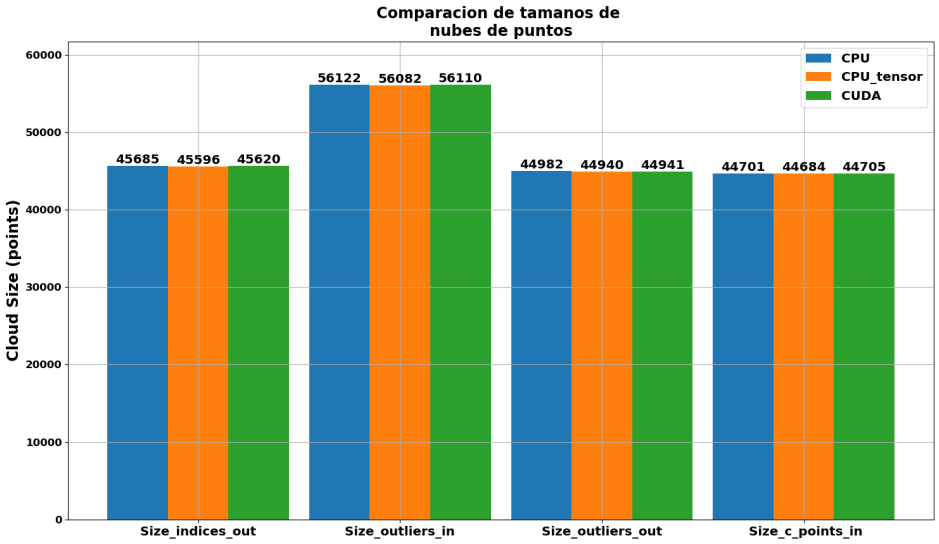

Figura 6.5 Tamaños de nubes con frustum doble - OMEN - Dron en el suelo.

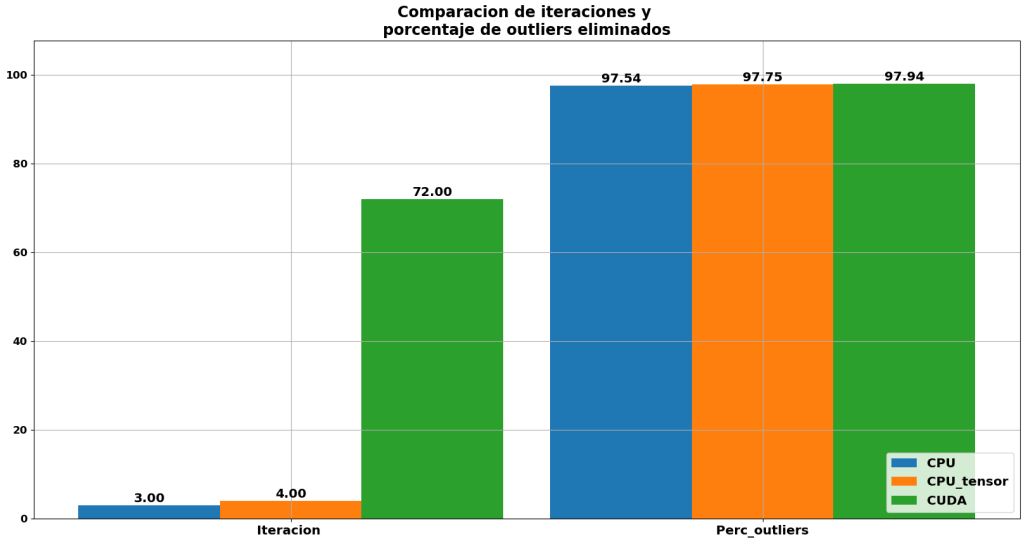

Figura 6.6 Iteraciones y porcentaje outliers con frustum doble - OMEN - Dron en el suelo.

6.2.2 Jetson TX2

Frustum individual

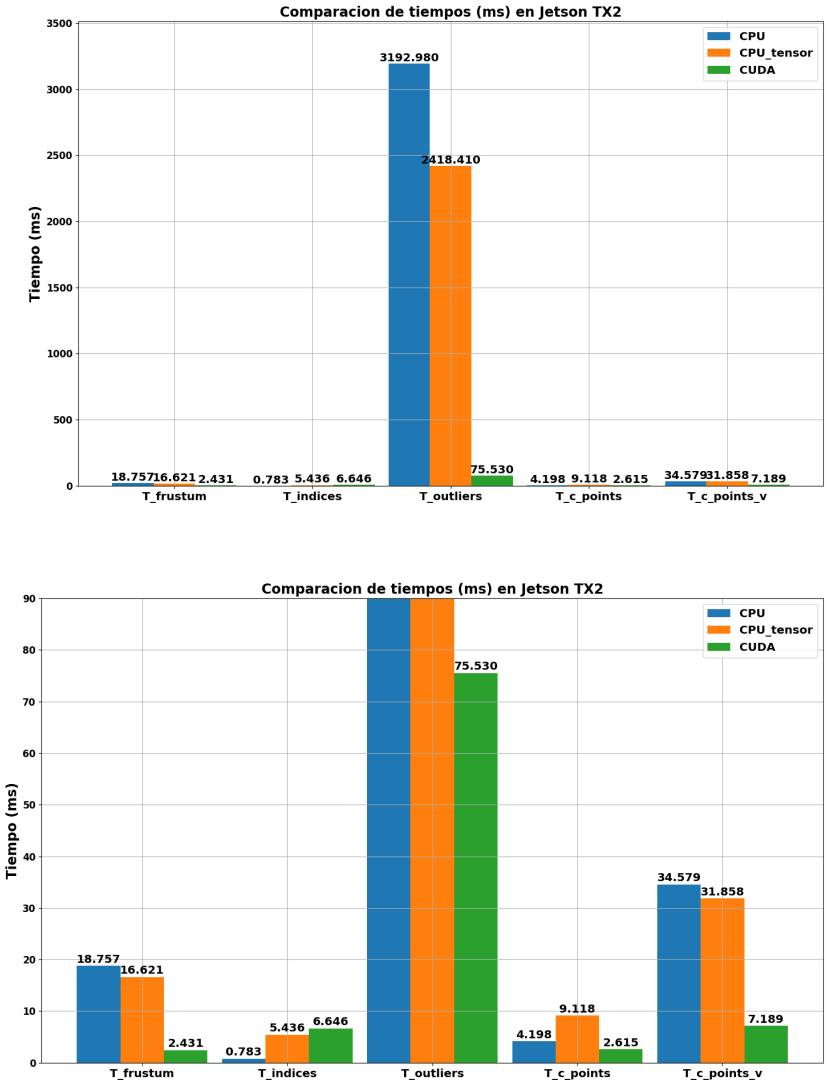

Figura 6.7 Tiempos de ejecución con frustum individual - Jetson TX2 - Dron en el suelo.

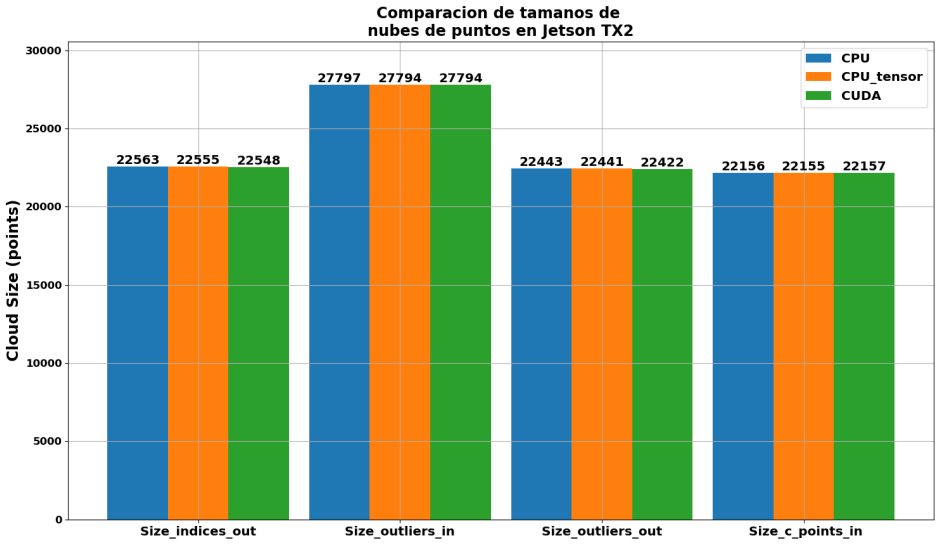

Figura 6.8 Tamaños de nubes con frustum individual - Jetson TX2 - Dron en el suelo.

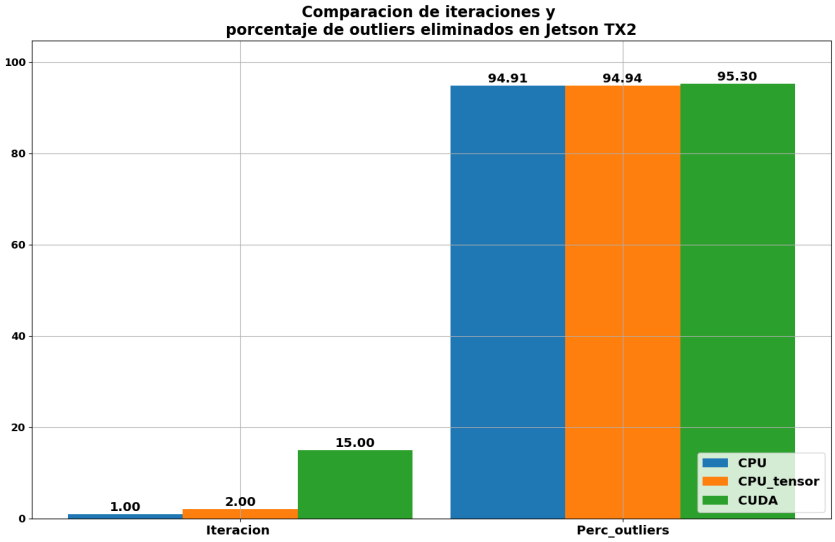

Figura 6.9 Iteraciones y porcentaje outliers con frustum individual - Jetson TX2 - Dron en el suelo.

Frustum doble

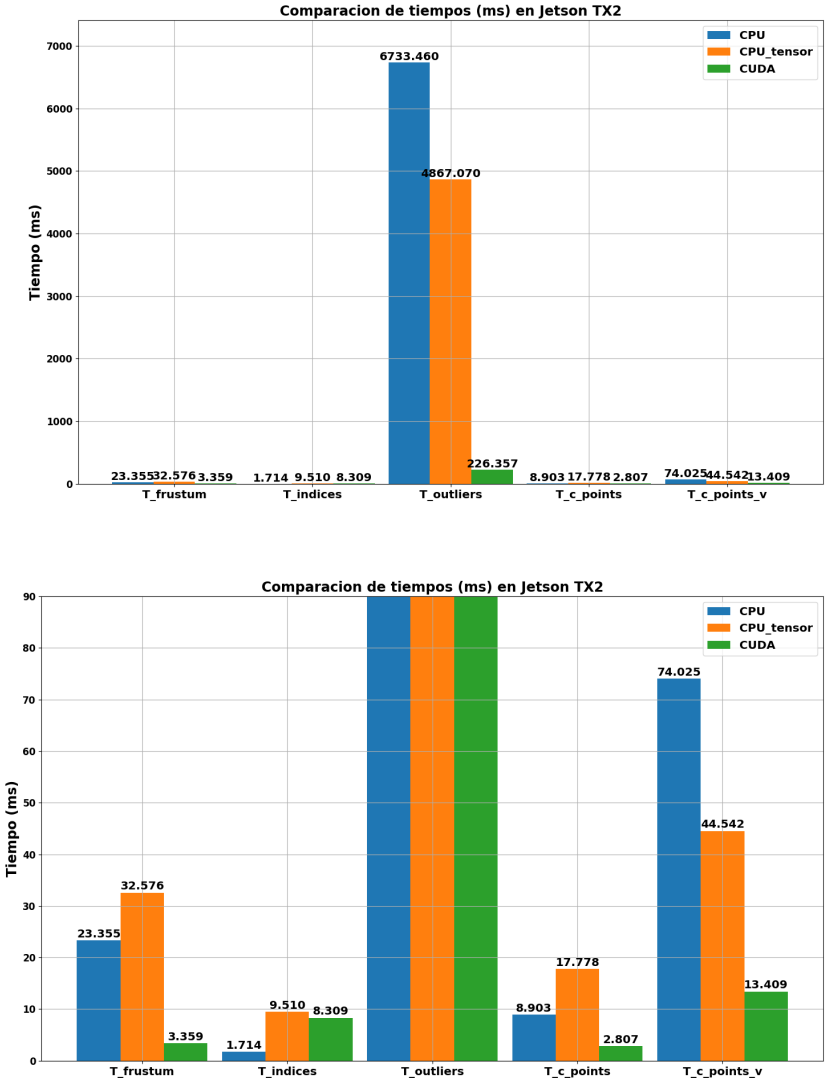

Figura 6.10 Tiempos de ejecución con frustum doble - Jetson TX2 - Dron en el suelo.

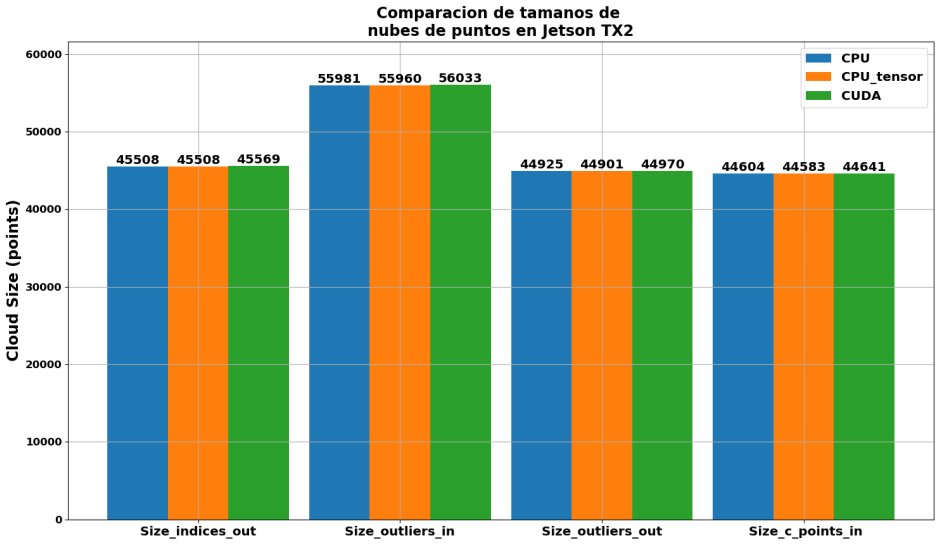

Figura 6.11 Tamaños de nubes con frustum doble - Jetson TX2 - Dron en el suelo.

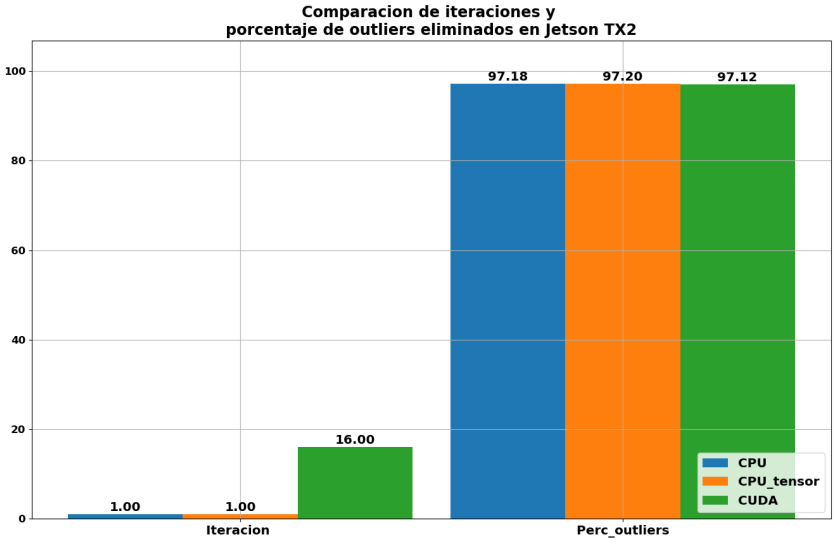

Figura 6.12 Iteraciones y porcentaje outliers con frustum doble - Jetson TX2 - Dron en el suelo.

6.3 Aterrizaje

Para este segundo escenario se decidió sacar métricas únicamente con el filtro frustum individual.

6.3.1 OMEN

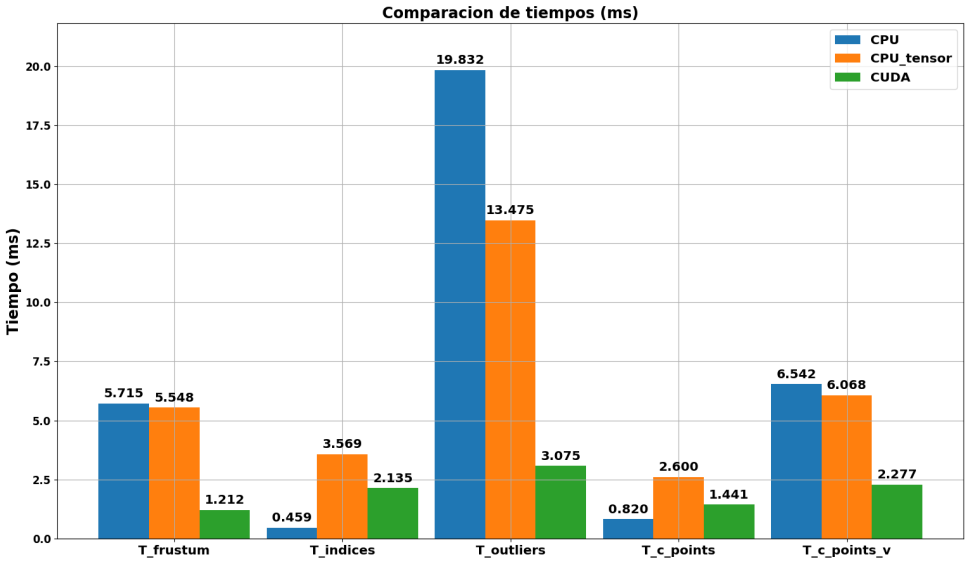

Figura 6.13 Tiempos de ejecución con frustum individual - OMEN - Aterrizaje.

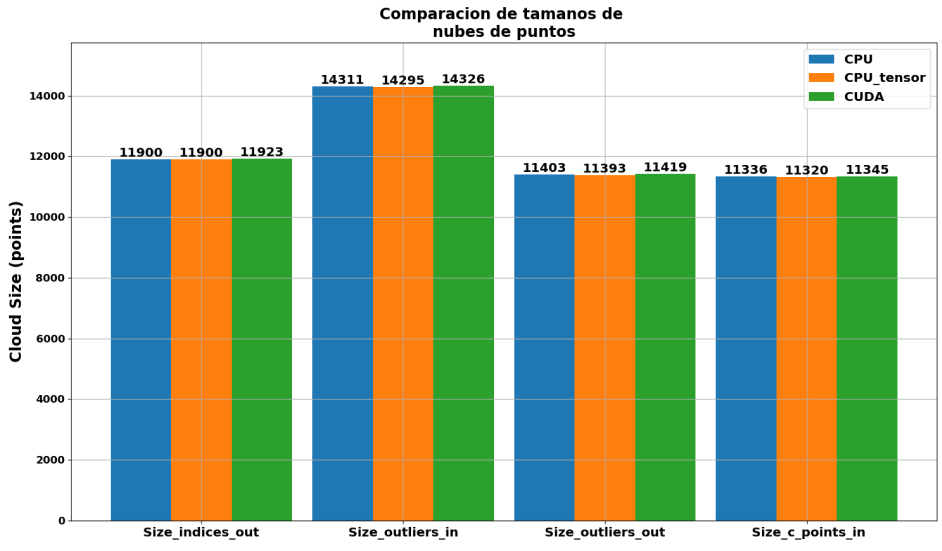

Figura 6.14 Tamaños de nubes con frustum individual - OMEN - Aterrizaje.

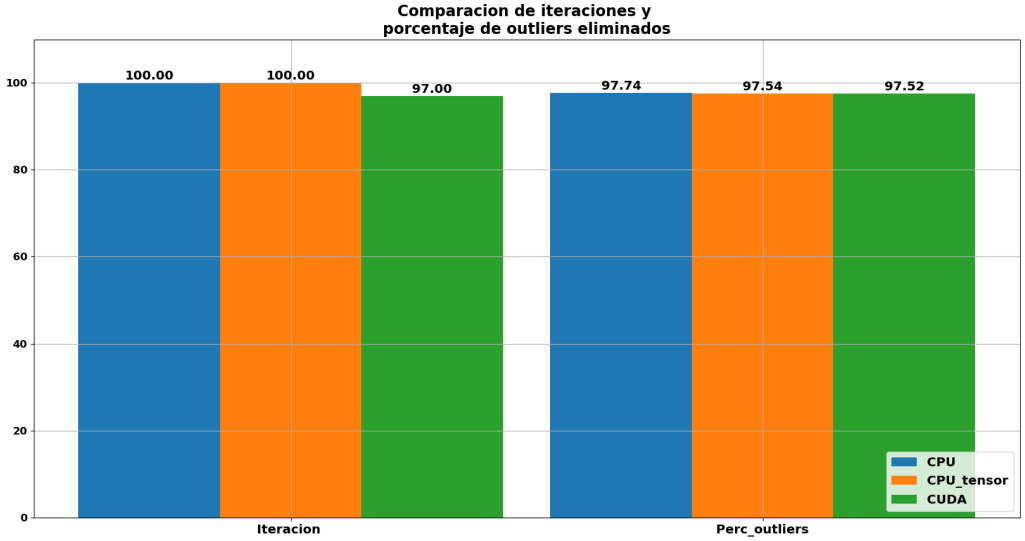

Figura 6.15 Iteraciones y porcentaje outliers con frustum individual - OMEN - Aterrizaje.

6.3.2 Jetson TX2

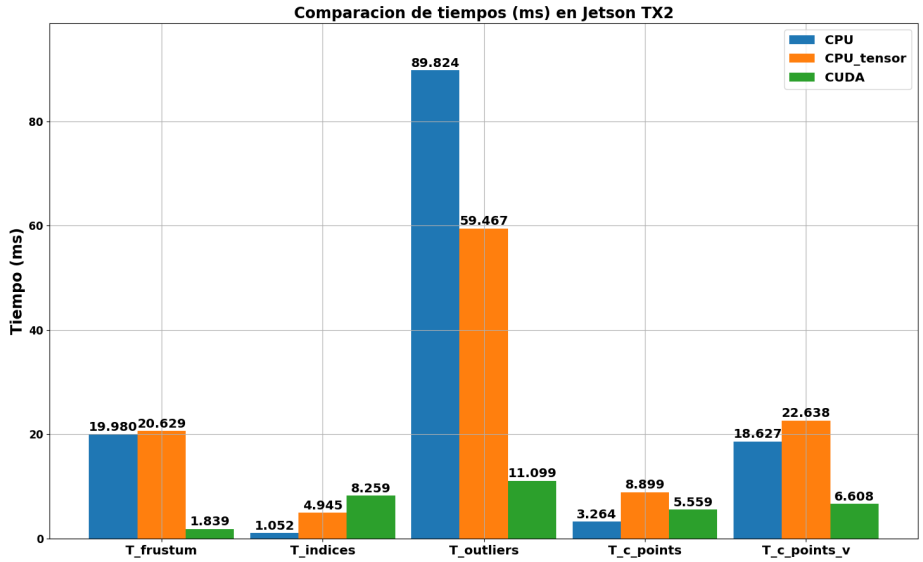

Figura 6.16 Tiempos de ejecución con frustum individual - Jetson TX2 - Aterrizaje.

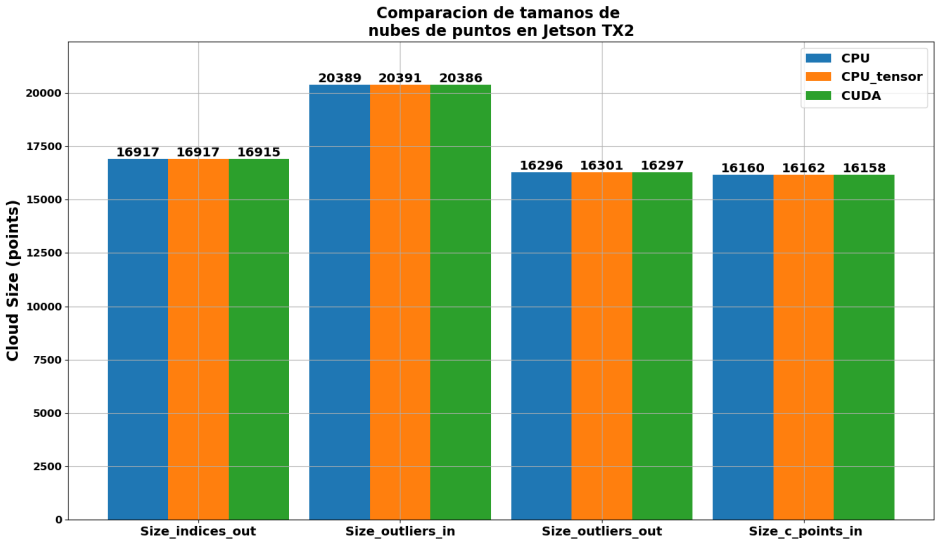

Figura 6.17 Tamaños de nubes con frustum individual - Jetson TX2 - Aterrizaje.

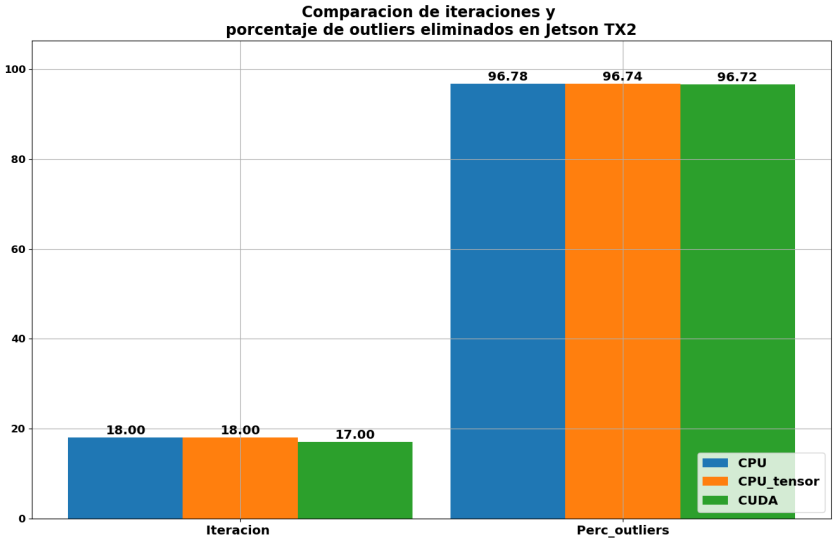

Figura 6.18 Iteraciones y porcentaje outliers con frustum individual - Jetson TX2 - Aterrizaje.

6.4 Nube de puntos pequeña y dispersa

6.4.1 OMEN

Frustum individual

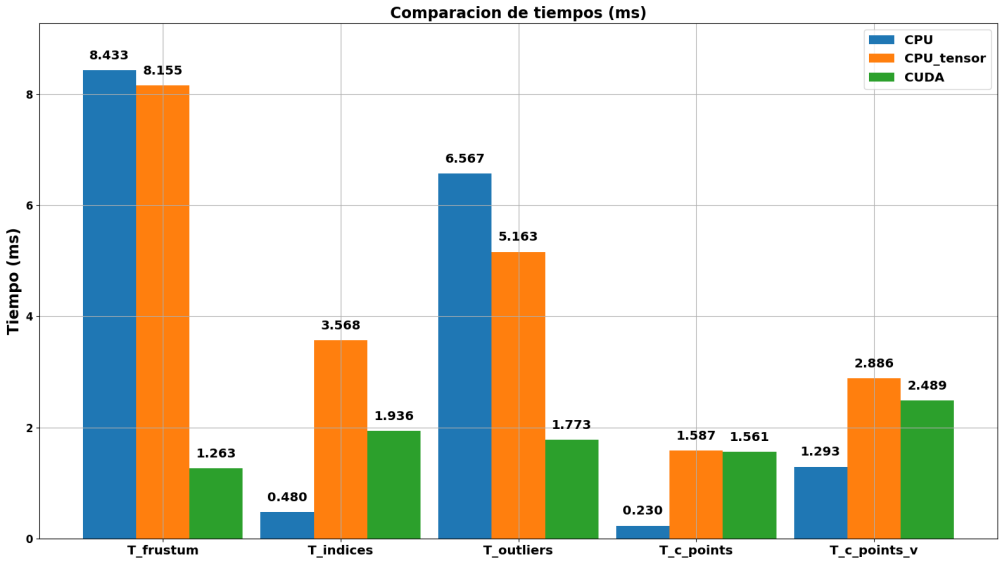

Figura 6.19 Tiempos de ejecución con frustum individual - OMEN - Nube pequeña.

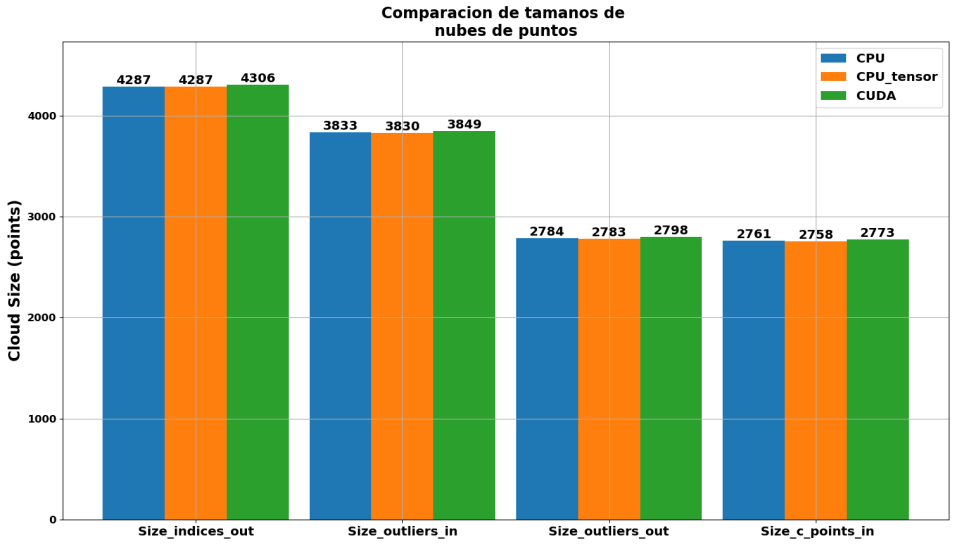

Figura 6.20 Tamaños de nubes con frustum individual - OMEN - Nube pequeña.

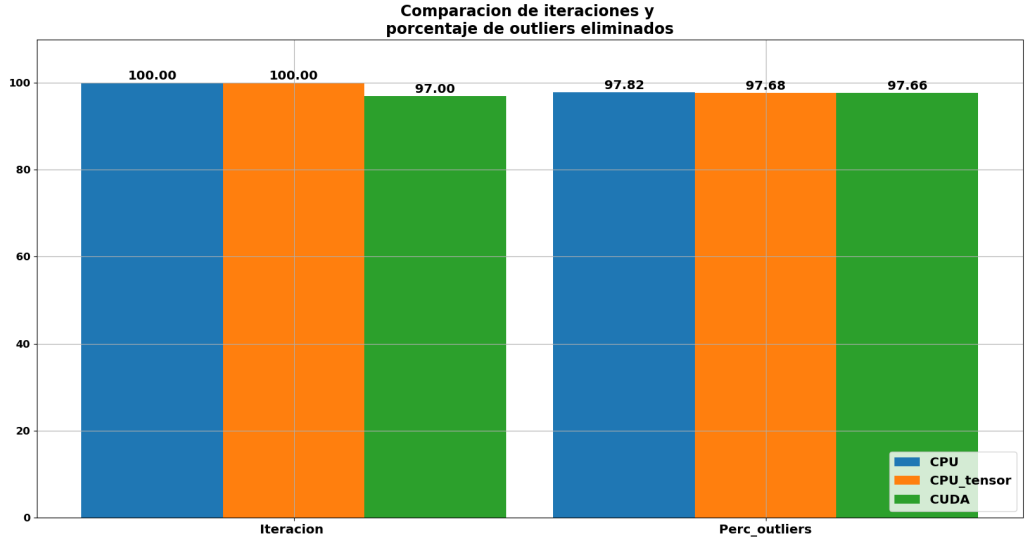

Figura 6.21 Iteraciones y porcentaje outliers con frustum individual - OMEN - Nube pequeña.

Frustum doble

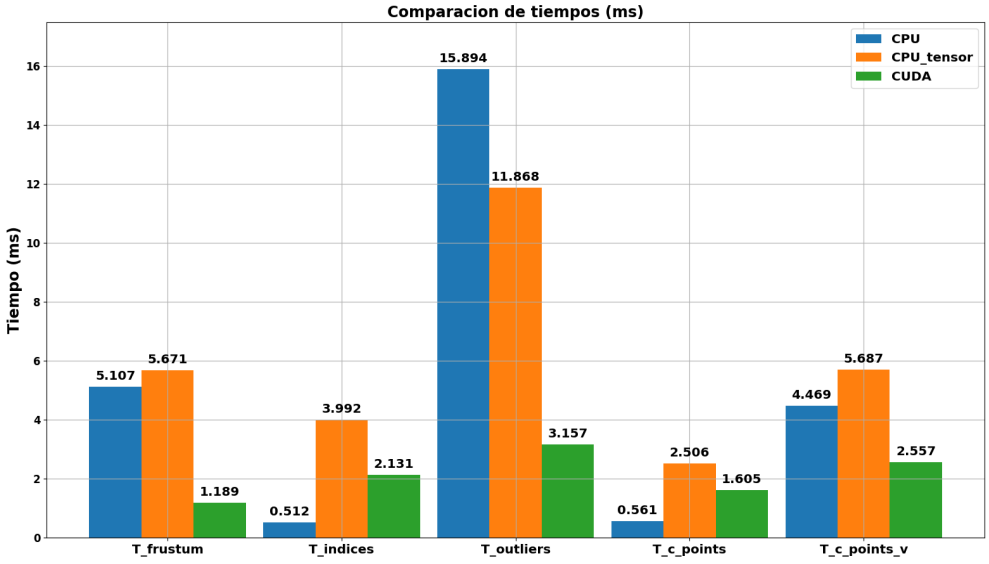

Figura 6.22 Tiempos de ejecución con frustum doble - OMEN - Nube pequeña.

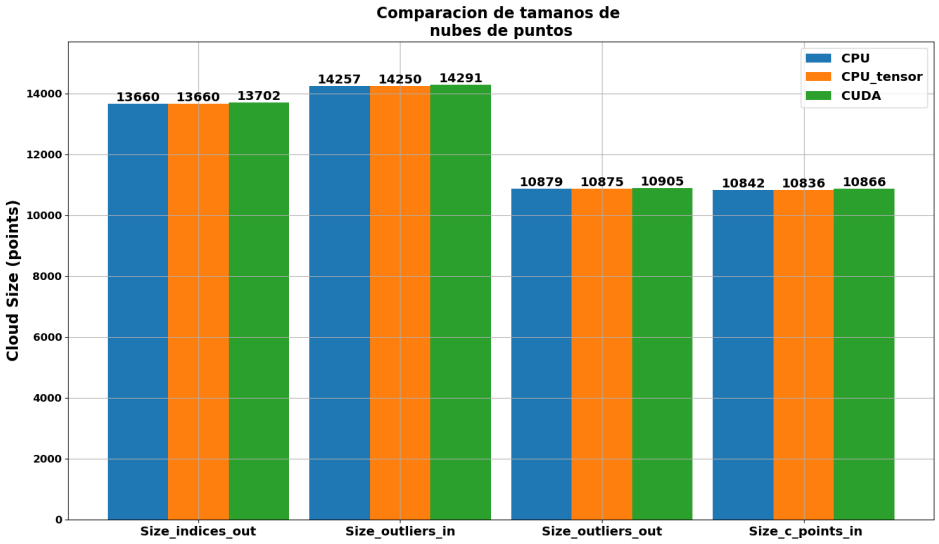

Figura 6.23 Tamaños de nubes con frustum doble - OMEN - Nube pequeña.

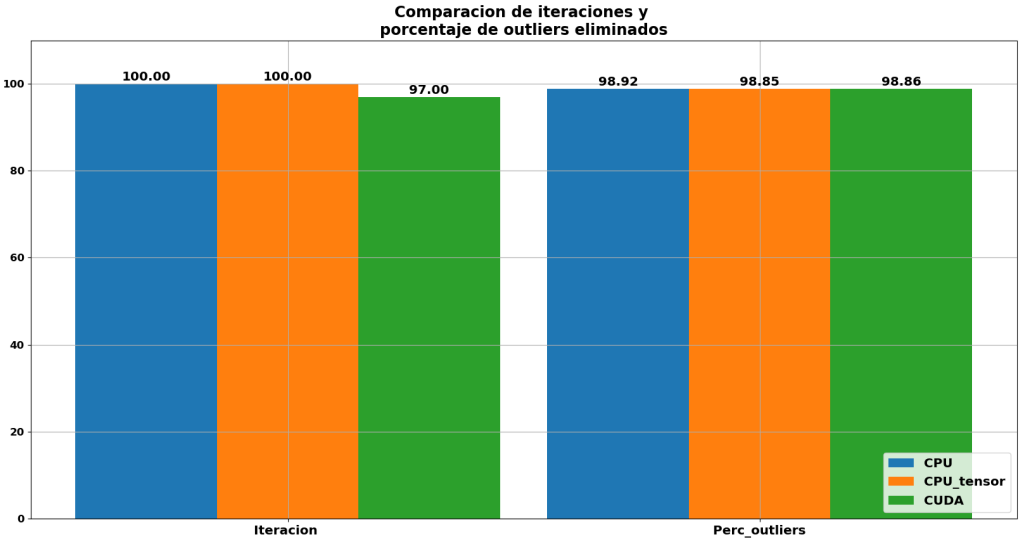

Figura 6.24 Iteraciones y porcentaje outliers con frustum doble - OMEN - Nube pequeña.

6.4.2 Jetson TX2

Frustum individual

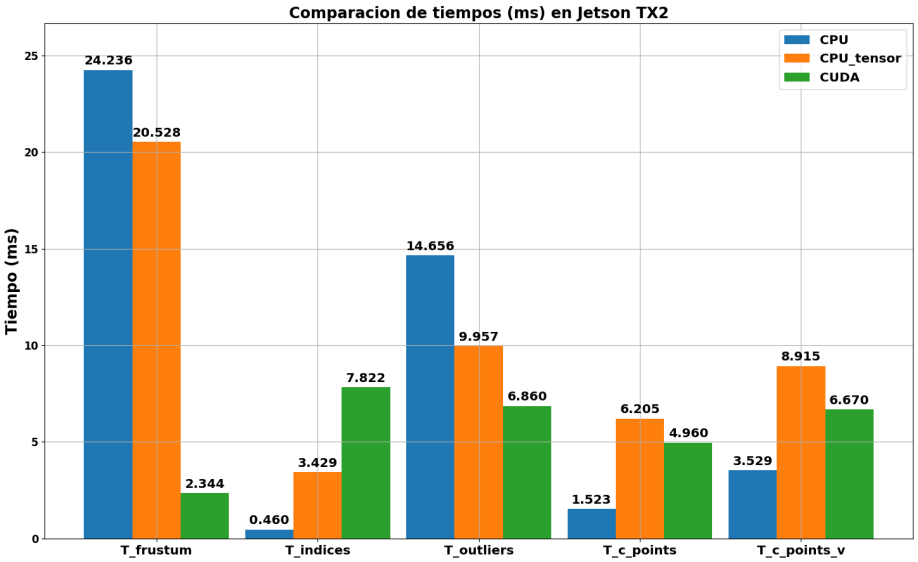

Figura 6.25 Tiempos de ejecución con frustum individual - Jetson TX2 - Nube pequeña.

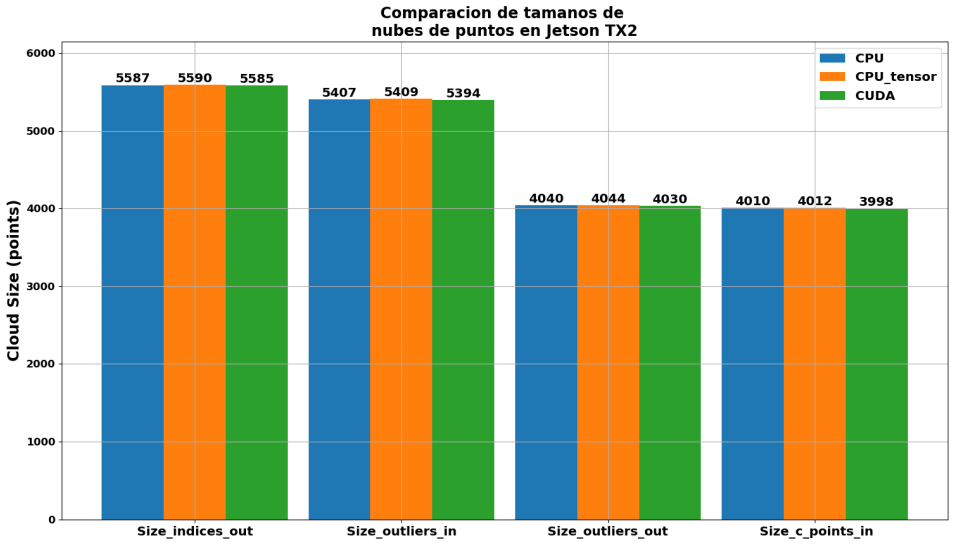

Figura 6.26 Tamaños de nubes con frustum individual - Jetson TX2 - Nube pequeña.

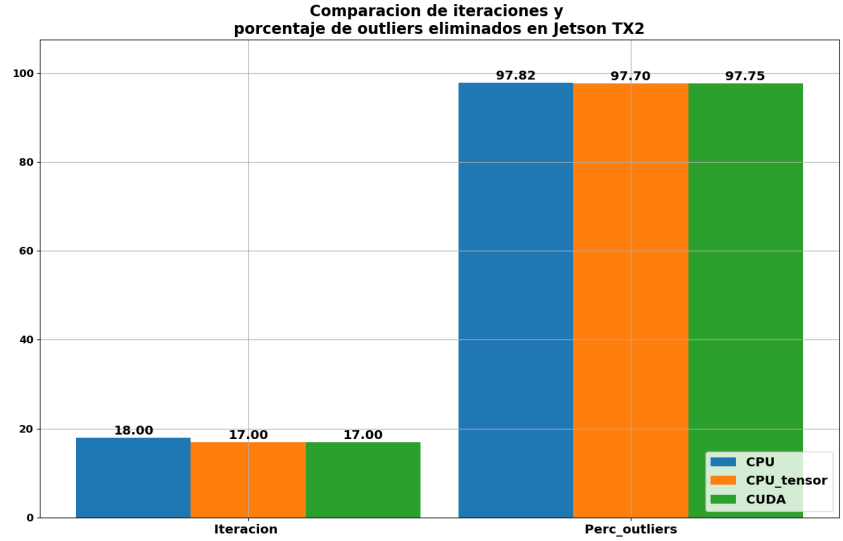

Figura 6.27 Iteraciones y porcentaje outliers con frustum individual - Jetson TX2 - Nube pequeña.

Frustum doble

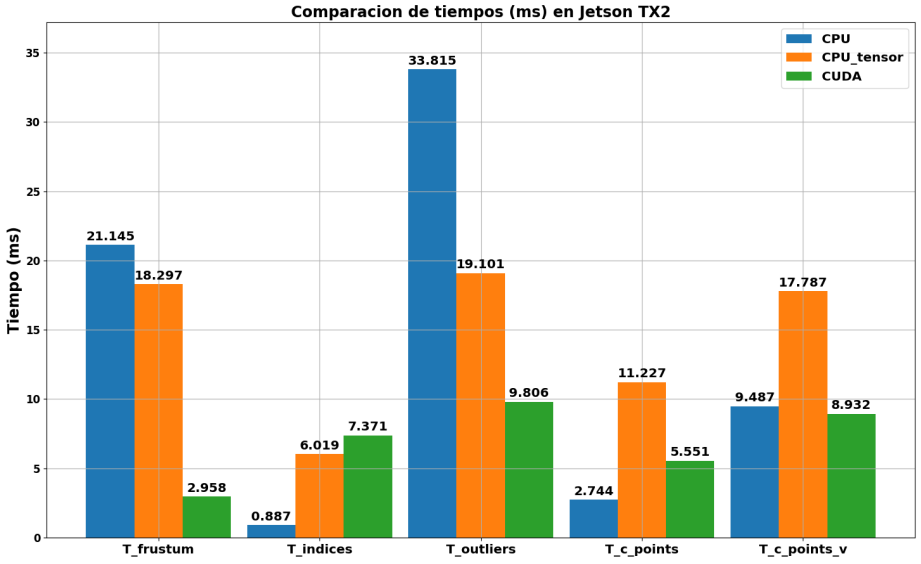

Figura 6.28 Tiempos de ejecución con frustum doble - Jetson TX2 - Nube pequeña.

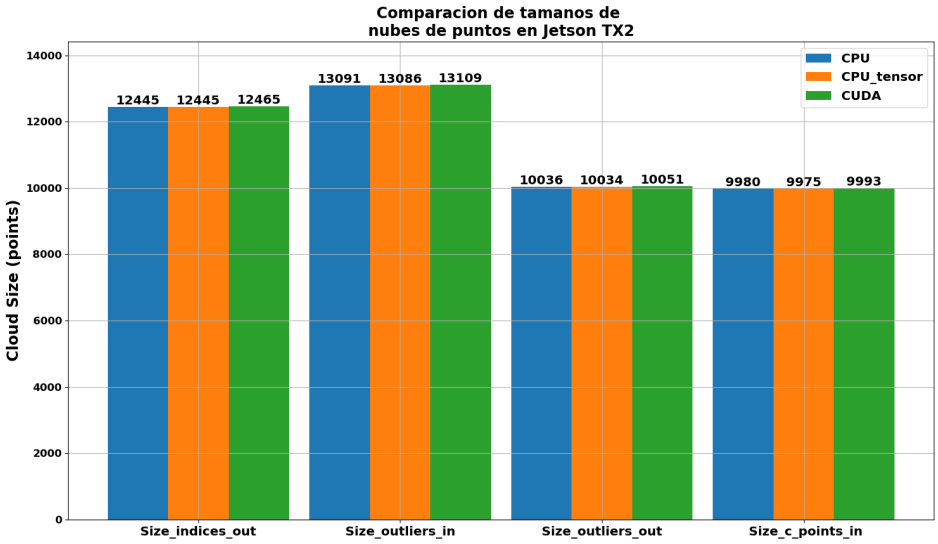

Figura 6.29 Tamaños de nubes con frustum doble - Jetson TX2 - Nube pequeña.

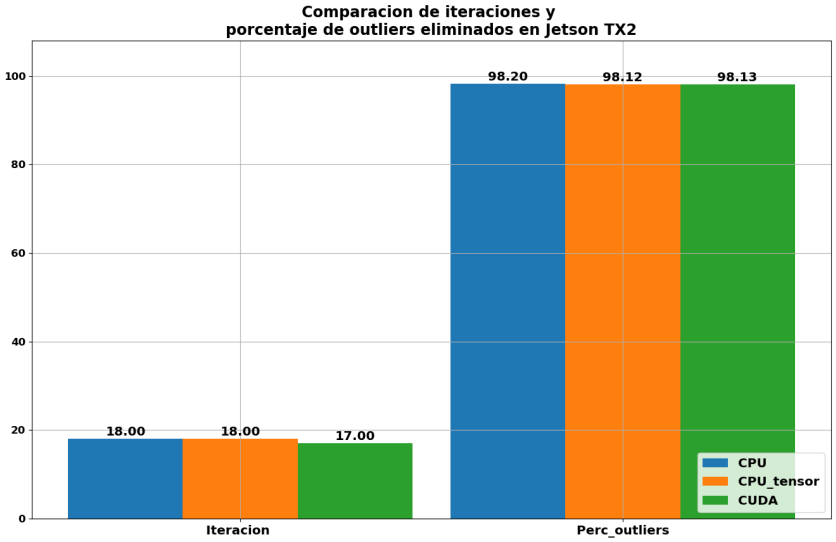

Figura 6.30 Iteraciones y porcentaje outliers con frustum doble - Jetson TX2 - Nube pequeña.

6.5 Nube de puntos intermedia

En este último escenario también se decidió obtener métricas sólo con el filtro frustum individual.

6.5.1 OMEN

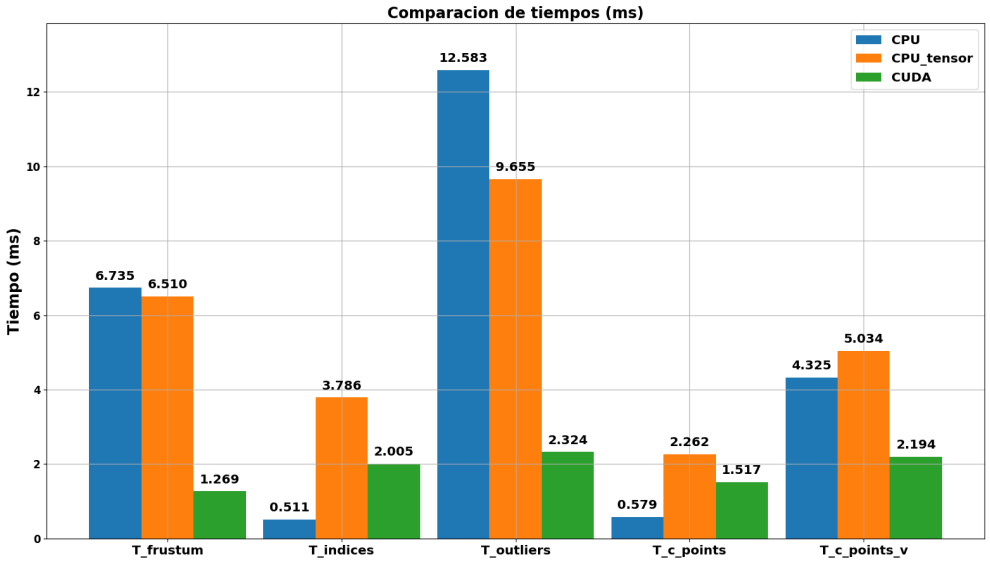

Figura 6.31 Tiempos de ejecución con frustum individual - OMEN - Nube intermedia.

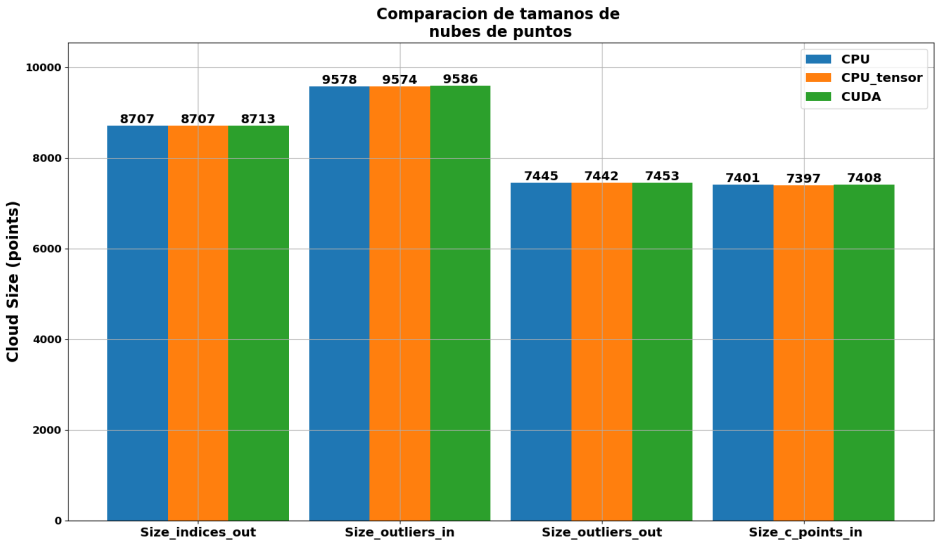

Figura 6.32 Tamaños de nubes con frustum individual - OMEN - Nube intermedia.

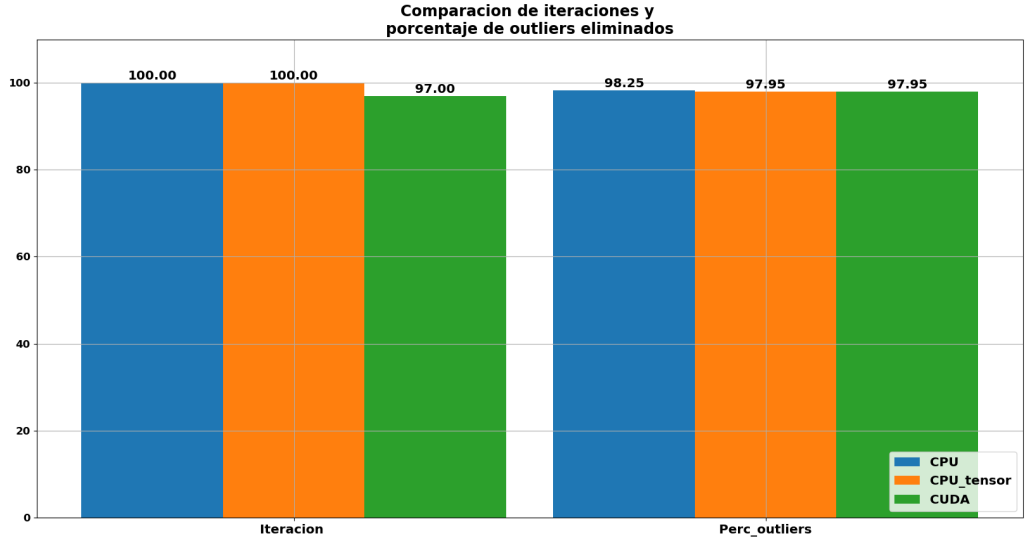

Figura 6.33 Iteraciones y porcentaje outliers con frustum individual - OMEN - Nube intermedia.

6.5.2 Jetson TX2

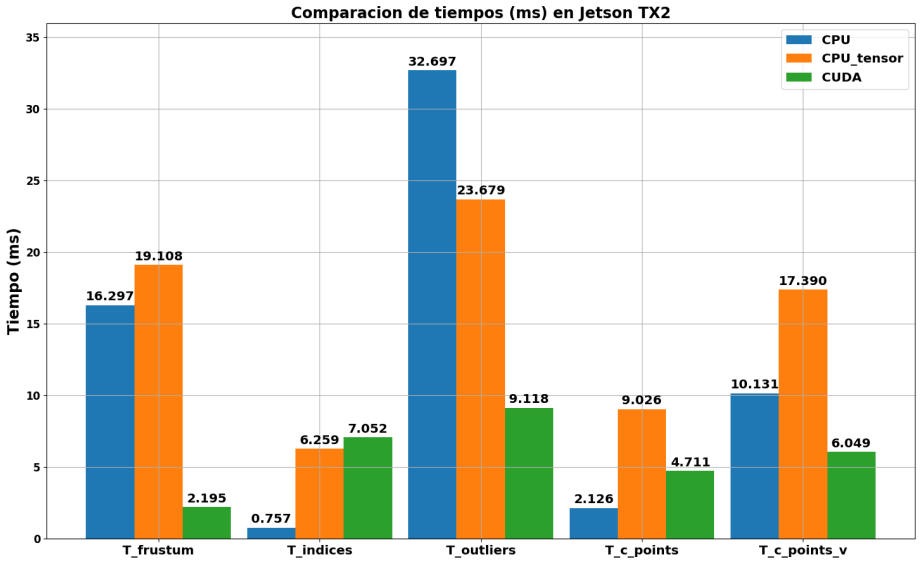

Figura 6.34 Tiempos de ejecución con frustum individual - Jetson TX2 - Nube intermedia.

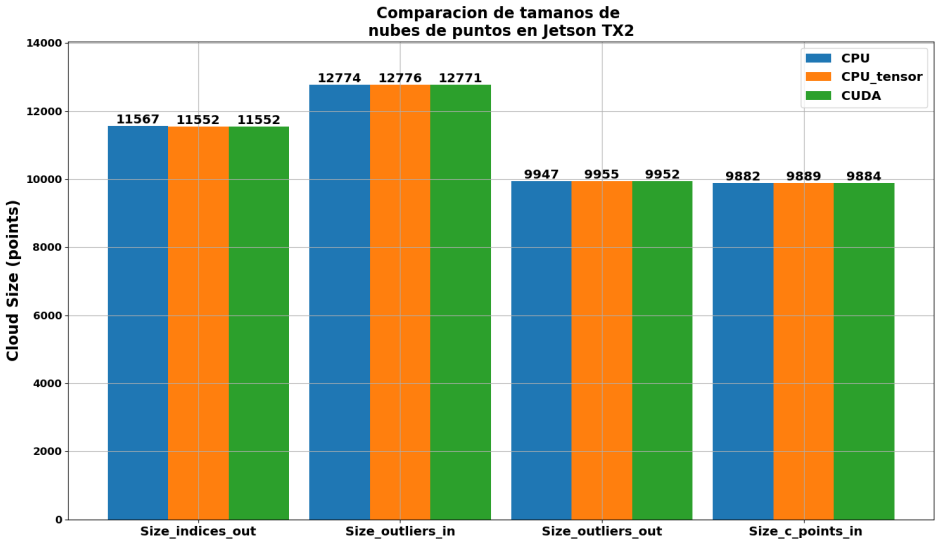

Figura 6.35 Tamaños de nubes con frustum individual - Jetson TX2 - Nube intermedia.

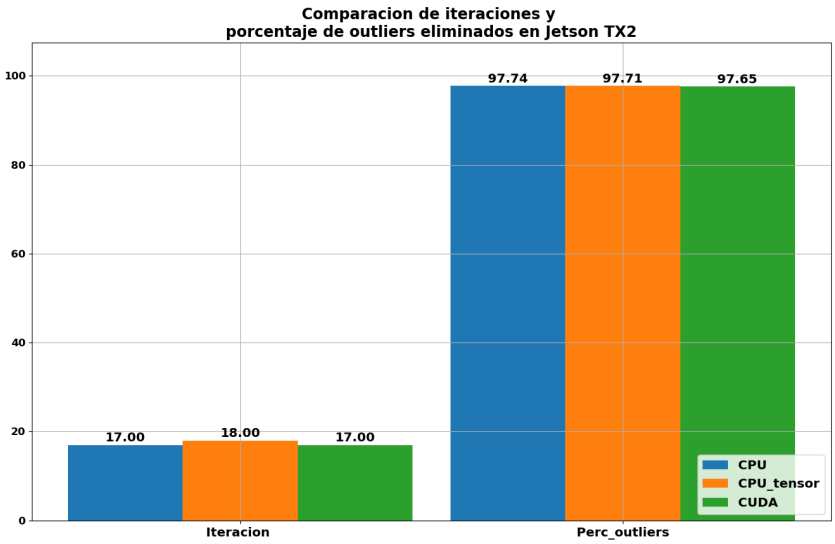

Figura 6.36 Iteraciones y porcentaje outliers con frustum individual - Jetson TX2 - Nube intermedia.

6.6 Análisis de los resultados

Tras visualizar todas estas gráficas de métricas obtenidas podemos realizar un análisis al completo de los resultados. En términos generales, es fácil de observar que cuanto más grandes sean los tamaños de las nubes de puntos, mayores serán los tiempos de ejecución, lo cual puede provocar una disminución en el número de iteraciones ejecutadas, y los porcentajes de *outliers* eliminados también se verían afectados cayendo ligeramente. Un aspecto que llama la atención son los elevados tiempos de ejecución del algoritmo de *extracción de outliers*, especialmente en el escenario del *dron en el suelo*, pues aparte de que la nube de puntos sea muy grande, ésta también es muy densa en las proximidades inmediatas del dron por estar el *LIDAR* tan cercano al suelo; esta concentración de puntos provoca que el algoritmo de búsqueda de vecinos tenga una enorme *carga computacional* y su tiempo de respuesta sea inviable para el procesamiento de nubes de puntos desde un dron. A continuación, se analizarán con más detalles los 4 escenarios para ambos dispositivos, el portátil *OMEN* y la *Jetson TX2*, tanto con filtro frustum individual como doble para ver la importancia de los tamaños de las nubes de puntos.

Empezando por la ejecución en el portátil OMEN con un filtro frustum individual, se observa que se han realizado 7 iteraciones usando CPU, 15 para CPU con tensores y 88 para CUDA en el escenario del suelo mientras que en el resto de escenarios se han producido 100 iteraciones usando CPU y CPU con tensores y 97 con CUDA, la reducción drástica de iteraciones en el escenario del suelo viene motivada por lo explicado anteriormente, y hace notar la mejora que supone trabajar con CUDA en dicho escenario manteniendo un número elevado de iteraciones a pesar de la carga computacional a la que se ve expuesto el algoritmo. Respecto a tiempos de ejecución, el *filtro frustum* presenta unos tiempos casi 3 veces más rápidos usando CUDA que CPU en el escenario del suelo y 5 veces en el resto de escenarios; el *extractor de índices* es más rápido usando CPU, mientras que los tiempos con CUDA son unas 4 veces más lentos y con CPU con tensores unas 8 veces; el *extractor de outliers* tiene unos tiempos de ejecución con CUDA 6 veces más rápidos que en la CPU (60 veces más rápido en el escenario del suelo) y 4 veces más rápidos que la CPU con tensores (30 veces en el escenario del suelo) y los porcentajes de *outliers* eliminados son similares (ligeramente inferior en el escenario del suelo); en el *extractor de puntos más cercanos sin vecindad* la CPU es más rápida que CUDA (unas 7 veces más en el escenario de nube pequeña y unas 2 veces más en el escenario de aterrizaje) y que la CPU con tensores (7 y 3 veces más en los mismos escenarios); y finalmente el mismo algoritmo pero con *vecindad* presenta unos tiempos de ejecución unas 2 veces más rápidos que la CPU con tensores y CUDA en el escenario de la nube pequeña mientras que en el resto CUDA es mucho más rápido que la CPU con y sin tensores.

En el caso de aplicar un frustum doble en el portátil OMEN, se realizan 100 iteraciones en la CPU con y sin tensores y 97 en CUDA en el escenario de nube pequeña mientras que en el de suelo se realiza un número de iteraciones muy inferior, 3 con CPU, 4 con CPU con tensores y 72 con CUDA. En cuanto a los algoritmos, el filtro *frustum* es 5 veces más rápido usando CUDA que la CPU con tensores en el escenario de nube pequeña y unas 8 veces en el del suelo; el *extractor de índices* resulta más eficiente ejecutarlo en la CPU pues es 4 veces más rápido que con CUDA y unas 8 veces con la CPU con tensores; el *extractor de outliers* tiene unos tiempos de ejecución con CUDA 5 veces más rápidos que en la CPU y 4 veces con la CPU con tensores en el escenario de nube pequeña, mientras que en el del suelo se obtienen unos tiempos bastante elevados siendo la ejecución en CUDA 40 y 26 veces más rápida que en la CPU sin y con tensores respectivamente, en cambio los porcentajes de *outliers* eliminados son similares; en el *extractor de puntos más cercanos sin vecindad* la CPU es 3 veces más rápida que CUDA y 5 más que la CPU con tensores en el escenario de la nube pequeña mientras que en el del suelo CUDA es 2 y 10 veces más rápida que la CPU sin y con tensores; y en el mismo algoritmo pero con vecindad activada los tiempos con CUDA son unas 2 veces más rápidos que con la CPU sin y con tensores.

Por otro lado, en la *Jetson TX2* aplicando un filtro *frustum* individual se realizan unas 17-18 iteraciones para los 3 tipos de almacenamiento en los escenarios de aterrizaje, nube intermedia y nube pequeña, mientras que en el escenario del suelo se realizan 1, 2 y 15 iteraciones en CPU, CPU con tensores y CUDA respectivamente. Con respecto al funcionamiento de los algoritmos, en el filtro *frustum* la ejecución con CUDA es 8 veces más rápida que en la CPU con tensores en el escenario del suelo y 10 veces en el resto de escenarios; en el *extractor de índices*, la CPU es entre 5 y 8 veces más rápida que la CPU con tensores y entre 8 y 15 veces más que en la GPU según el escenario (a menor tamaño de nube, más rápida es la CPU

con respecto a las otras); el *extractor de outliers* tiene unos tiempos de ejecución con CUDA entre unas 2 y 42 veces más rápidos que usando la CPU sin y con tensores según el escenario, pues a mayor tamaño de la nube mayor es la eficiencia del uso de CUDA respecto a las otras dos opciones, además los porcentajes de *outliers* eliminados son similares (ligeramente inferior en el escenario del suelo); en el *extractor de puntos más cercanos sin vecindad* la CPU es entre 2 y 3 veces más rápida que CUDA y entre 3 y 5 veces más que la CPU con tensores en los escenarios del aterrizaje y las nubes pequeña e intermedia, mientras que en el del suelo CUDA es 2 y 4 veces más rápida que la CPU sin y con tensores; y en el mismo extractor pero activando la *vecindad* la CPU es unas 2 veces más rápida que CUDA y unas 3 más que la CPU con tensores en el escenario de la nube pequeña, mientras que en el resto de escenarios la ejecución en CUDA es entre 2 y 5 veces más rápida que la CPU y entre 3 y 4 veces más que la CPU con tensores.

Finalmente, en la *Jetson TX2* aplicando un filtro *frustum* doble se realizan unas 17-18 iteraciones para los 3 tipos de almacenamiento en el escenario de la nube pequeña, mientras que en el escenario del suelo se realizan 1 y 16 iteraciones en las dos CPU y CUDA respectivamente. El filtro *frustum* es unas 7 veces más rápido en la GPU que en la CPU y unas 6-10 (nube pequeña-suelo) veces más que la CPU con tensores; el *extractor de índices* presenta unos tiempos de ejecución de unas 5-8 (nube pequeña-suelo) veces más rápidos que CUDA y 6-7 veces más que CPU con tensores; en el *extractor de outliers* los tiempos con CUDA son unas 3-30 (nube pequeña-suelo) veces más rápidos en CUDA que en CPU y 2-22 veces más que la CPU con tensores; el *extractor de puntos más cercanos sin vecindad* posee unos tiempos de ejecución en la CPU 2 y 4 veces más rápidos que CUDA y CPU con tensores en el escenario de la nube pequeña, mientras que en el del suelo CUDA es 3 y 6 veces más rápido que la CPU sin y con tensores; si a este algoritmo se le indica vecindad los tiempos de ejecución en CUDA son entre 1 y 5 veces más eficaces que la CPU y entre 2 y 3 veces más que la CPU con tensores.

Para ilustrar y resumir este análisis tan denso sobre la eficiencia de CUDA frente a la CPU y la CPU con tensores se incluye la Tabla 6.2.

Tabla 6.2 Eficiencia de CUDA frente a CPU y CPU con tensores.

Dispositivo		**OMEN**		**JETSON**	
Filtro frustum		**Frustum indv**	**Frustum doble**	**Frustum indv**	**Frustum doble**
Eficiencia CUDA vs CPU	**Frustum**	-	-	-	-
	Indices	x 0.25/0.25/0.2/0.2	x 0.25/ - / - /0.3	x 0.06/0.1/0.13/0.11	x 0.12/ - / - /0.2
	Outliers	x 3.7/5.4/6.5/53	x 5/ - / - /40	x 2/4/8/42	x 3.4/ - / - /30
nube_p/nube_i/ aterr/suelo	**Closest**	x 0.15/0.4/0.6/0.4	x 0.3/ - / - /2	x 0.3/0.4/0.6/1.6	x 0.5/ - / - /3.1
	Closest_v	x 0.5/2/2.9/2	x 1.8/ - / - /2.7	x 0.5/1.6/2.8/4.8	x 1.1/ - / - /5.5
Eficiencia CUDA vs CPU_tensor nube_p/nube_i/ aterr/suelo	**Frustum**	x 6.5/5.1/4.6/2.8	x 5/ - / - /8	x 10/8.7/11/7	x 6/ - / - /9.6
	Indices	x 1.8/1.9/1.7/1.5	x 2/ - / - /3	x 0.4/0.9/0.6/0.8	x 0.8/ - / - /1.2
	Outliers	x 2.9/4.2/4.4/24	x 4/ - / - /26	x 1.4/2.6/5.3/32	x 2/ - / - /22
	Closest	x 1/1.5/1.8/1.6	x 1.5/ - / - /10	x 1.3/1.9/1.6/3.5	x 2/ - / - /6.4
	Closest_v	x 1.1/2.3/2.7/1.9	x 2.3/ - / - /2.7	x 1.3/2.9/3.4/4.4	x 2/ - / - /3.3
Iteraciones CPU/ CPU_tensor/ CUDA	**nube_p**	100 / 100 / 97	100 / 100 / 97	18 / 17 / 17	18 / 18 / 17
	nube_i	100 / 100 / 97	-	17 / 18 / 17	-
	aterrizaje	100 / 100 / 97	-	18 / 18 / 17	-
	suelo	7 / 15 / 88	7 / 15 / 88	1 / 2 / 15	1 / 1 / 16
% outliers eliminados	**nube_p**	98	99	98	98
	nube_i	98	-	98	-
	aterrizaje	98	-	97	-
	suelo	96	98	95	97

7 Conclusión y trabajo futuro

A modo de conclusión, del detallado análisis del Capítulo 6 podemos deducir que ante escenarios con nubes de puntos de gran tamaño que requieran de algoritmos de una elevada carga computacional, el uso de la GPU mediante las implementaciones de las funcionalidades con CUDA supone una gran mejora en la eficiencia del proceso reduciendo bastante los tiempos de ejecución. Por el contrario, en escenarios con nubes de puntos pequeñas o que simplemente el algoritmo que se vaya a ejecutar no requiera mucha carga computacional, las implementaciones con CUDA no resultan tan eficientes en comparación con aquellas en la CPU, esto es debido a que, en estas situaciones, el tiempo necesario para la reserva y almacenamiento de los datos en la memoria de CUDA resulta ser mayor que el tiempo requerido para operar con los datos almacenados en la CPU.

Otra de las conclusiones que podemos señalar es que, como era de esperar, en la *Jetson TX2* se obtienen peores resultados en cuanto a tiempos de ejecución tanto en CPU como en GPU debido a un menor número de núcleos de NVIDIA CUDA (256 núcleos en la *Jetson TX2* frente a los 640 de la *GeForce GTX 1050 Mobile*) y también a un menor rendimiento de la CPU provocado por una menor velocidad de CPU (2 GHz frente a 2.8 GHz) y un menor número de núcleos de CPU (6 frente a 8). Debido a este aumento de tiempos de ejecución los valores relativos de la eficiencia de CUDA frente a CPU de la Tabla 6.2 son, en general, ligeramente menores, sin embargo, esto no significa que el uso de CUDA en dispositivos como la *Jetson TX2* sea menos recomendado sino todo lo contrario. El hecho de que dichos valores sean solo ligeramente inferiores implica que, ante un gran aumento de los tiempos de ejecución, la reducción de tiempos de respuesta que se logra obtener haciendo uso de CUDA es mayor en la *Jetson TX2* que en el portátil *OMEN*. Por poner unos ejemplos, en la Figura 6.4 se observa que, para el escenario de dron en el suelo con frustum doble en el portátil *OMEN*, el *extractor de outliers* tarda unos 2220 ms en la CPU frente a unos 55 ms con CUDA, obteniendo así una diferencia de 2165 ms, y que el *extractor de puntos más cercanos con vecindad* tarda unos 20 ms en CPU y 7.5 ms en CUDA, obteniendo una reducción de 12.5 ms; mientras que en la Figura 6.10 se observa que en la *Jetson TX2* se obtienen unas diferencias de tiempos de 6507 ms en el *extractor de outliers* y 61 ms en el *extractor de puntos más cercanos con vecindad*.

En vistas de futuro y dados los resultados aquí mencionados, se ha establecido, mediante la realización de este proyecto, el camino a seguir para, a partir de ahora, empezar a adaptar los algoritmos de procesamiento de nubes de puntos en sistemas autónomos, a los que se le deberá de dotar de una GPU, de forma que se puedan ejecutar operaciones de alto coste computacional sin problema alguno en cuanto a retrasos o rendimiento. Ahora que he adquirido un nivel de experiencia considerable en este campo, podría volver a evaluar los errores obtenidos de las implementaciones de los algoritmos de *PCL* con *CUDA* para así evitar la necesidad de instalar una librería tan pesada como lo es *Open3D*. Sin embargo, opino que *Open3D* tiene un gran margen de crecimiento y mejora y no descarto que en un futuro no muy lejano ocupe el puesto que ostenta ahora mismo *PCL* en el área de procesamiento de datos 3D para operaciones en sistemas autónomos.

Bibliografía

[1] J. Zhang and S. Singh, "LOAM : Lidar Odometry and Mapping in real-time," in *Robotics: Science and Systems Conference (RSS)*, Jan 2014. [Online]. Available: https://www.researchgate.net/publication/311570125_LOAM_Lidar_Odometry_and_Mapping_in_real-time

[2] T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti, and D. Rus, "LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2020, pp. 5135–5142. [Online]. Available: https://doi.org/10.1109/IROS45743.2020.9341176

[3] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *2011 IEEE International Conference on Robotics and Automation*, May 2011. [Online]. Available: https://doi.org/10.1109/ICRA.2011.5980567

[4] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," Jan 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1801.09847

[5] (2022, Feb) Unidad de procesamiento gráfico. Wikipedia. [Online]. Available: https://es.wikipedia.org/wiki/Unidad_de_procesamiento_gráfico

[6] Modern GPU technology powers traditional graphics applications—and much more. Intel Corporation. [Online]. Available: https://www.intel.es/content/www/es/es/products/docs/processors/what-is-a-gpu.html

[7] S. Seth. (2022, Apr) GPU Usage in Cryptocurrency Mining. [Online]. Available: https://www.investopedia.com/tech/gpu-cryptocurrency-mining/

[8] M. Levinas. (2020, Nov) What Is GPU Computing And How Is It Applied Today? [Online]. Available: https://www.cherryservers.com/blog/what-is-gpu-computing

[9] (2022, Jun) Tegra. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Tegra

[10] (2022, Jun) GPUs supported by CUDA. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/CUDA#GPUs_supported

[11] P. Sakdhnagool. (2021, Oct) CUDA Tutorials. CUDA. [Online]. Available: https://cuda-tutorial.readthedocs.io/en/latest/

[12] M. Harris. (2013, Jan) Using Shared Memory in CUDA C/C++. [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[13] FindCUDAToolkit. CMake. [Online]. Available: https://cmake.org/cmake/help/latest/module/FindCUDAToolkit.html

[14] (2022, May) NVIDIA CUDA Compiler Driver NVCC. NVIDIA. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[15] pcl Namespace Reference. Point Cloud Library. [Online]. Available: https://pointclouds.org/documentation/namespacepcl.html

[16] open3d Namespace Reference C++. Open3D. [Online]. Available: http://www.open3d.org/docs/release/cpp_api/namespaceopen3d.html

[17] R. Singh. (2021, May) Tensor PointCloud IsSimilar, SelectByIndex, RemoveRadiusOutlier functions. Open3D. [Online]. Available: https://github.com/isl-org/Open3D/pull/3422

[18] Structures Having Eigen Members. Eigen. [Online]. Available: http://eigen.tuxfamily.org/dox-devel/group__TopicStructHavingEigenMembers.html

[19] P. Jiménez. (2021, Nov) EigenMatrixToTensor allows both row-major and col-major. Open3D. [Online]. Available: https://github.com/isl-org/Open3D/issues/4261

[20] Y. Lao. (2022, Jan) core::EigenMatrixToTensor: allow convertion for eigen vector. Open3D. [Online]. Available: https://github.com/isl-org/Open3D/pull/4579

[21] P. Stotko. (2021, Sep) Random Tensor Maker. Open3D. [Online]. Available: https://github.com/isl-org/Open3D/blob/5e047b812182a63d67d2ca515ee0625d7c56ceb1/cpp/benchmarks/benchmark_utilities/Rand.cpp

[22] NVIDIA GeForce GTX 1050 Mobile Specifications. TECHPOWERUP. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-1050-mobile.c2917

[23] NVIDIA Jetson TX2 Specifications. TECHPOWERUP. [Online]. Available: https://www.techpowerup.com/gpu-specs/jetson-tx2-gpu.c3231