Departamento de Ingeniería de Computadores
Facultad de Informática de A Coruña
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE MÁSTER
MÁSTER INTERUNIVERSITARIO EN
COMPUTACIÓN DE ALTAS PRESTACIONES

Mapeo de imágenes multiespectrales sobre nubes de puntos 3D de alta densidad en GPU

Estudiante: Juan Manuel Jurado Rodríguez
Directores: Emilio José Padrón González
Francisco Ramón Feito Higueruela

A Coruña, 1 de septiembre de 2020.

Agradecimientos

Para la realización de este trabajo me gustaría agradecer el pleno apoyo de los tutores que me han brindado su ayuda desde el comienzo hasta el final de este proyecto. Gracias a ellos he aprendido nuevos conceptos y buenas prácticas entorno a la programación paralela, GPGPU y HPC.

Resumen

En este proyecto se propone un método para el *mapeo* de imágenes multiespectrales sobre extensas nubes de puntos basado en GPU. En primer lugar, se lleva a cabo una revisión de las estructuras de datos, rendimiento y código del método secuencial del que se dispone al inicio de este trabajo. Del mismo modo, también se estudian las características más relevantes del dataset, compuesto por una nube de puntos de alta resolución y una galería de imágenes multiespectrales. En segundo lugar, se desarrolla una paralelización del método en OpenMP con el fin de testear el rendimiento ejecutando múltiples hilos paralelos en la CPU. En tercer lugar, se lleva a cabo una fase de disgregación del método original para su adecuada ejecución en la GPU. Se propone así, un nuevo enfoque en el que se tiene como objetivo minimizar el número de transferencias de memoria de CPU a GPU, así como permitir el procesado de nubes de puntos arbitrariamente grandes. Como resultado se propone una solución out-of-core, desarrollada en CUDA, que no es dependiente de ningún hardware específico en el marco de los procesadores NVIDIA. Por último, se realiza una evaluación del rendimiento de secuencial y en paralelo para obtener la aceleración global del método.

Abstract

In this work, we propose a GPU-based method for multispectral image mapping on high dense point clouds. Firstly, we carried out a review of the available sequential method considering the used data structures, performance and code. Likewise, we analysed the main features of the study dataset, which was formed by an RGB point cloud and many multispectral images. Secondly, we explored an OpenMP-based approach in order to test the performance launching multiple parallel threads on CPU. Thirdly, we proposed a new workflow of the sequential method in order to be launched efficiently on GPU. Consequently, a novel approach was proposed in which we focused on the optimization of data transfers between CPU and GPU as well as to process enormous point cloud for image mapping and occlusion test. As a result, an out-of-core solution was developed using CUDA, which was not dependent to a specific graphics hardware of NVIDIA. Finally, the performance of parallel and sequential method was measured in order to obtain the global acceleration of the proposed method.

Palabras chave:

GPGPU
Computación paralela
Nube de puntos
Mapeo 3D
Imagen multiespectral

Keywords:

GPGPU
Parallel computing
Point cloud
3D mapping
Multispectral imaging

Índice general

1 Introducción **1**

2 Estado del arte **3**

3 Estudio del método y datos **7**

3.1 Descripción del conjunto de datos 7

3.2 Descripción del algoritmo 9

3.2.1 Preprocesamiento 10

3.2.2 Inicialización de estructuras 10

3.2.3 Mapeo 3D 11

3.2.4 Oclusión 13

3.2.5 Programación OpenMP 14

3.2.6 Limitaciones del enfoque secuencial 15

4 Optimización del método en GPU **17**

4.1 Introducción 17

4.1.1 GPGPU 17

4.1.2 Fundamentos de la Programación Paralela en GPU 18

4.1.3 Motivación 19

4.2 Arquitectura del Sistema 20

4.3 Desarrollo del Método en Paralelo 20

4.3.1 Primera versión (v.1) 21

4.3.2 Segunda versión (v.2) 22

4.3.3 Tercera versión (v.3) 24

5 Evaluación del rendimiento **27**

5.1 Ejecución secuencial 27

5.2 Paralelización en CPU 29

5.3 Paralelización en GPU 30

6 Conclusiones **33**

A Glosario de acrónimos **37**

B Glosario de terminos **39**

Bibliografía **41**

Índice de figuras

3.1 Nube de puntos 3D. 7
3.2 Imágenes multiespectrales: (**a**) Banda del verde, (**b**) Banda del infrarrojo cercano, (**c**) Banda del rojo y; (**c**) Banda de bordes rojos. 8
3.3 3D canvas: (**a**) Cono de visión para una cámara multiespectral; y (**b**)Visualización de cámaras multiespectral y nube de puntos en la escena. 10
3.4 Conjunto de estructuras de datos principales utilizadas en el método. 11
3.5 Proyección de un punto sobre una imagen con deformación fisheye. 13
3.6 Proyección de un conjunto de puntos 3D sobre una cámara. 13

4.1 Chip GV100 de la arquitectura Volta 18
4.2 CUDA: Modelo de programación. 19
4.3 Arquitectura del sistema. 20
4.4 Esquema ilustrado sobre la computación en el *kernel* de mapeo 3D 21
4.5 Solución *out-of-core* (v.3.1). 24
4.6 Solución *out-of-core* (v.3.2). 25

5.1 Comparativa de rendimiento para cada operación (mapeo, oclusión) y total en secuencial y paralelización en CPU (OpenMP). 30
5.2 Comparativa de tiempos globales sobre el conjunto de entrenamiento: (1) secuencial, (2) OpenMP y (3) CUDA. 32
5.3 Comparativa de tiempos globales sobre *dataset* completo: (1) secuencial, (2) OpenMP y (3) CUDA. 32

Índice de tablas

5.1 Niveles de optimización del compilador (los niveles son incrementales: cada uno de ellos incluye las optimizaciones de los niveles anteriores). 28
5.2 Medición de tiempos de cómputo (ms) con el compilador gcc y icc^{TM}. 28
5.3 Medición de tiempos de cómputo (ms) con OpenMP. 29
5.4 Medición de tiempos de cómputo (ms) comparando las versiones v.2 y v.3.1 (out-of-core síncrona). 31
5.5 Tiempo global de ejecución de la versión v.2 y v.3.2 en ms 31

Capítulo 1

Introducción

En el panorama tecnológico actual, el desarrollo de aplicaciones basadas en HPC (Computación de Altas Prestaciones o High Performance Computing en inglés) es indispensable para permitir el procesamiento y cálculo de grandes volúmenes de datos que son extraídos del mundo real. Junto a lo anterior, los recientes avances en la producción de nuevos sensores capaces de capturar distintos fenómenos físicos tales como la reflectividad en distintas longitudes de onda o la apariencia que presentan los objetos posibilita la adquisición de datos heterogéneos descriptivos de nuestro entorno. Además, mediante el uso de cámaras de alta resolución o sistemas LiDAR (Light Detection and Ranging), se puede modelar la geometría de los objetos capturados con el fin de ser representados en un escenario tridimensional (3D). No cabe duda de que el uso de modelos 3D para la simulación de cualquier escenario del mundo real facilita la tarea de inspección visual y análisis. Sin embargo, la generación de modelos 3D enriquecidos con varias capas temáticas de información que describan sus principales características no es trivial. En este ámbito de investigación se ubica el presente proyecto caracterizado por su carácter multidisciplinar en el que se aúna el conocimiento de distintas áreas como la computación de altas prestaciones, informática gráfica y teledetección.

El tipo de información a adquirir sobre cualquier escenario de nuestro entorno puede llegar a ser muy diverso e incluso según la tipología de cada entorno los parámetros de estudio varían significativamente. Por ello, el caso de uso sobre el que se ha trabajado en este proyecto se centra en el estudio de espacios naturales. En concreto, el *dataset* utilizado está formado por imágenes aéreas RGB y multiespectrales tomadas desde un dron. De este modo, se tiene un conjunto de datos heterogéneos que ha sido adquirido mediante distintos sensores, una cámara digital de alta y un sensor multiespectral.

En este proyecto se toma como punto de partida el trabajo presentado por Jurado et al. [1] en el que se propone un método para el *mapeo* de información multiespectral en una nube de puntos. El principal objetivo de este trabajo es proponer una variación del método original de tal forma que se puedan manejar nubes de puntos más extensas y se optimice el tiempo de

cómputo haciendo uso de un enfoque basado en GPU (Graphics Processing Unit).

Tras este capítulo de introducción, el contenido de la memoria queda estructurado de la siguiente forma: en el Capítulo 2 se expone el estado del arte en el que se exponen otros trabajos relacionados, en el Capítulo 3 se realiza el estudio del método secuencial, el conjunto de datos y se desarrolló una versión paralela del método en CPU, en el Capítulo 4 se expone el desarrollo del método acelerado en GPU, en el Capítulo 5 se realizó la evaluación del rendimiento y, por último, en el Capítulo 6 se muestran las principales conclusiones y trabajo futuro.

Capítulo 2

Estado del arte

El continuo desarrollo de sensores dirigidos a la observación de nuestro entorno potenciado por el despliegue de nuevos sistemas de adquisición basados en drones, tiene como consecuencia la generación de un conocimiento de gran interés para multitud de propósitos científicos [2, 3]. En el marco de este proyecto, uno de los retos actuales está relacionado con la fusión de datos heterogéneos, adquiridos por diferentes sensores, en el mismo sistema de referencia. Del mismo modo, se plantean una serie de limitaciones inherentes a la explotación de modelos 3D debido a la carga computacional asociada al manejo y visualización de los mismos. Por consiguiente, junto al avance tecnológico de nuevos sensores con los que generar extensos *datasets*, se requiere el desarrollo de metodologías y algoritmos para la gestión y procesamiento eficiente de los datos.

Recientemente se han lanzado al mercado una amplia variedad de sensores y cámaras de alta resolución basadas en drones con la finalidad de monitorizar entornos naturales con un alto nivel de detalle. Centrándonos en el ámbito de trabajo de este proyecto, la observación de espacios naturales lleva consigo una complejidad añadida debido a la superposición de elementos en el área de estudio, la irregularidad geométrica de los objetos y el carácter dinámico de los materiales que coexisten en él. No obstante, gracias al uso de sistemas dron es posible la toma de imágenes desde numerosos puntos de vista con el fin de capturar el entorno con la mayor completitud posible. Para generar los modelos 3D del mundo real se pueden utilizar sistemas LiDAR [4] que proporcionan nubes de puntos 3D y los resultados en escenarios naturales son bastante satisfactorios al ser capaces de generar puntos 3D tras hojas y vegetación no muy densa. Sin embargo, estos sensores tienen un precio elevado en el mercado por lo que existen otras soluciones software basadas en imágenes. El conocido método structure-from-motion (SfM) [5] es ampliamente utilizado para la generación de nubes de puntos y mallas de triángulos en 3D a partir de imágenes superpuestas entre sí. Junto con el aumento de la resolución de las imágenes, los modelos 3D resultantes alcanzan un mayor detalle geométrico y en la mayoría de los casos representan fielmente las formas de los objetos capturados. La

nube de puntos es el tipo de dato más utilizado para la representación 3D de árboles y plantas reconstruidos del mundo real. Si se pretende crear modelos volumétricos en este tipo de entornos es habitual la aplicación de métodos procedurales inversos considerando la nube de puntos como guía para la generación procedural de la geometría. No obstante, en este proyecto se trabajan directamente con nubes de puntos para evitar añadir complejidad extra y testear la eficiencia de los métodos propuestos.

Una vez modelada la geometría del entorno el siguiente paso es conocer las características de los materiales existentes atendiendo a su signatura espectral en distintas longitudes de onda. Para ello, se utilizan sensores multiespectral e hiperespectrales que capturan la irradiancia incidente y reflejada sobre la superficie de los objetos. En este proyecto se va a trabajar con un dataset generado por un sensor multiespectral (modelo: Parrot Sequoia) que captura imágenes en cuatro rangos espectrales: green, near-infrared, red y red-edge. Hasta la fecha, han sido un gran número de trabajos que se sirven de estos datos multiespectrales para estudiar la variabilidad espectral en entornos naturales [6], segmentación de materiales [7], caracterización espectral de distintas especies arbóreas [8], estudios sobre respuestas espectrales ante distintos fenómenos físicos [9], simulaciones [10], entre otros. No obstante, la fusión de datos espectrales y modelos 3D de tal forma que se mapee sobre la propia geometría los mapas de reflectancia plantea una serie de dificultades que son abordadas en este proyecto. En años anteriores, varios trabajos que abordan este objetivo fueron presentados superándose hitos relevantes en esta área de estudio [11, 12, 13]. Todos ellos trabajan con nubes espectrales de baja resolución espacial lo que limita el estudio en zonas que no son reconstruidas en el modelo. En Jurado et al. [1] se propone un método para el mapeo de imágenes multiespectrales sobre nubes reconstruidas a partir de imágenes RGB de alta resolución. De este modo, se consigue texturizar un modelo de una densidad de puntos en torno a 1.5 cm. con mapas de reflectancia. Este método abre posibilidades de estudio sobre la correlación existente entre características geométricas y rasgos espectrales en entornos naturales. No obstante, este método presenta una limitación significativa que tiene que ver con el rendimiento de las operaciones implementadas a medida que se trabajan con nubes de puntos de mayor tamaño. Para superar este problema, se estudia un enfoque basado en computación paralela utilizando la tarjeta gráfica.

El uso de procesadores gráficos para la computación de propósito general (GPGPU) ha favorecido la resolución de problemas complejos aprovechando las capacidades de cómputo de una GPU. Este tipo de procesadores fueron diseñados para la generación de gráficos 3D, sin embargo, junto al avance tecnológico que han experimentado su aplicación se ha extendido especialmente en el ámbito científico y de simulación [14, 15]. Las GPU modernas son procesadores de muchos núcleos totalmente programables que se caracterizan por su alta eficiencia energética y una buena relación precio-rendimiento. El desarrollo de algoritmos GPGPU tiene como base el estudio inicial del comportamiento del algoritmo en CPU, si lo hubiera, y

la adaptación del método para su ejecución en GPU optimizando los accesos de memoria y manejo de estructura de datos eficientes [16]. En esta linea de trabajo, en 2007, la compañía NVIDIA lanzó una plataforma de computación en paralelo denominada CUDA (Compute Unified Device Architecture) [17] para el desarrollo de algoritmos en GPUs de NVIDIA. Hasta la fecha, se han realizado numerosos estudios comparando el rendimiento entre varias librerías como CUDA [18], OpenCL [19] y OpenACC [20]. En general, CUDA alcanza un incremento significativo del rendimiento frente a soluciones basadas en OpenACC o OpenCL [21, 22]. En el reciente evento GTC 2020, NVIDIA presentó una nueva arquitectura para sus tarjetas denominada Ampere y en concreto la integración de los procesadores A100 que ofrecen una aceleración hasta 6 veces más que su antecesor (V100) con un impacto directo en áreas tales como la inteligencia artificial, el análisis masivo de datos y soluciones HPC. Además, también se presentó la tercera generación de interconexión NVLink de alta velocidad lo que mejora significativamente la escalabilidad, el rendimiento y la confiabilidad del cómputo en paralelo utilizando múltiples GPUs. Todos estos avances tecnológicos tienen un impacto directo sobre el desarrollo de métodos para la aceleración en GPU del procesamiento y extracción de características de imágenes de alta resolución [23, 24] y mapeo de imágenes multiespectrales aéreas sobre ortomosaicos [25]. Hasta nuestro conocimiento, no hay ninguna solución basada en GPU para la creación de nube de puntos espectrales a partir del mapeo de imágenes por lo que el método que se presenta supone una innovación en campos de aplicación como la teledetección o la informática gráfica.

En este proyecto se ha trabajado en el desarrollo de un algoritmo para el mapeo de imágenes multiespectrales sobre nubes de puntos de gran tamaño haciendo uso de la capacidad de calculo que proporciona la GPU. Se trata así de un trabajo de investigación que se nutre de un método ya publicado por el propio autor [1] y sobre el que se propone un nuevo enfoque que optimiza operaciones costosas relativas al proceso de mapeo y oclusión. De esta forma, se pretende ampliar la capacidad del método para garantizar un adecuado rendimiento en el manejo de modelos 3D con un tamaño en torno a 40 millones de puntos. Como resultado de este trabajo se obtiene una nueva versión del método optimizada en el que se abordan una serie de dificultades derivadas de la adaptación del algoritmo a la GPU y de la propia tipología del problema planteado que son descritas en la presente memoria.

Capítulo 3

Estudio del método y datos

En este capítulo se detallan las características más relevantes en torno al conjunto de datos utilizado por el método de estudio, así como las principales fases llevadas a cabo para la fusión de datos espectrales en nubes de puntos 3D.

3.1 Descripción del conjunto de datos

El conjunto de datos utilizado para este trabajo ha sido adquirido haciendo uso del equipamiento tecnológico disponible en la Universidad de Jaén. Aunque el proceso de adquisición está fuera del ámbito de interés para este trabajo, se considera necesaria una breve descripción del conjunto de datos utilizados para entender las operaciones desarrolladas con ellos y la dimensionalidad de los mismos. El área de estudio es una zona forestal del bosque Mediterráneo en la que coexisten distintas especies árboles tales como encinas, eucaliptos y pinos. Se cubre un área de 0.6 hectáreas (ha) y se utilizó un dron en el que se acoplan dos sensores: (1) una cámara digital RGB y (2) un sensor multiespectral. Ambos dispositivos capturan imágenes de forma simultánea.

Figura 3.1: Nube de puntos 3D.

En primer lugar, a partir de las imágenes RGB de alta resolución se lleva a cabo un proceso de fotogrametría para generar una nube de puntos. La densidad de esta nube de puntos es de 2500 puntos por m^3 con un tamaño total de 73.246.870 puntos. A este modelo se le aplica un filtro para la eliminación de ruido reduciendo el número de puntos a 66.374.475. La Figura 3.1 muestra la nube de puntos resultante.

En segundo lugar, se toman un amplio conjunto de imágenes multiespectrales con el sensor Sequoia [26]. Este dispositivo está compuesto por cuatro lentes que capturan la luz reflejada en una determinada franja del espectro electromagnético. Estas bandas se denominan de la siguiente manera: verde (530 nm a 570 nm), rojo (640 nm a 680 nm), infrarrojo cercano (770 nm a 810 nm) y bordes rojos (730 nm a 740 nm). En la Figura 3.2 se muestra una imagen correspondiente a cada una de las bandas. Además, este sensor mide la cantidad de luz incidente por el sol, que es utilizada junto con la irradiancia reflejada para calcular un valor de reflectancia para cada píxel de la imagen. El dataset de estudio contiene un total de 180 imágenes.

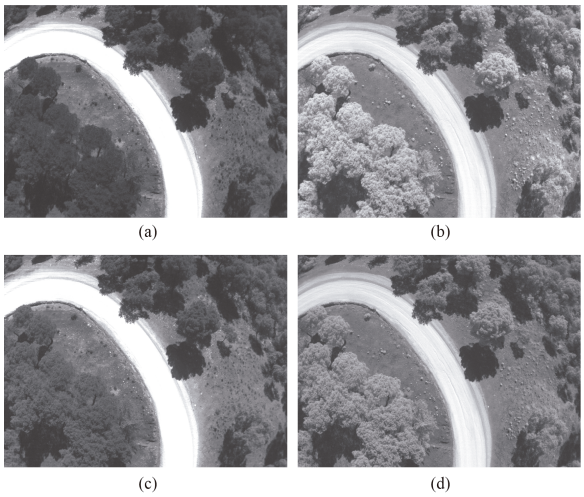

Figura 3.2: Imágenes multiespectrales: (**a**) Banda del verde, (**b**) Banda del infrarrojo cercano, (**c**) Banda del rojo y; (**c**) Banda de bordes rojos.

El tamaño de la nube de puntos y el número de imágenes a proyectar son factores determinantes que tienen un impacto sobre el rendimiento final del método. Además, aunque en menor medida, conforme se aumenta el número de imágenes multiespectrales se requiere de un tiempo mayor para la ejecución del método. Todo ello se describe en las siguientes secciones, pero previo a esto se procede a presentar el método estudiado.

3.2 Descripción del algoritmo

El algoritmo de estudio fue propuesto por J.M. Jurado et al. [1], autor de este trabajo. Este algoritmo desarrollado en secuencial sobre CPU tiene como finalidad principal el enriquecimiento de una nube de puntos con información espectral a partir del mapeo de imágenes multiespectrales. De esta forma, se pretende conocer sobre un modelo 3D las características relativas a las propiedades de los materiales existentes, lo que es de gran interés en áreas como informática gráfica y visión por computador. Estos valores espectrales son capturados por un sensor multiespectral capaz de medir la irradiancia incidente y reflejada con una alta resolución espacial en varias longitudes de onda. Toda esta información espectral queda registrada en imágenes que, tras un proceso de calibración radiométrica, son utilizadas como datos de entrada del método de estudio para proyectar el valor de reflectancia, medido por cada píxel, sobre la nube de puntos. A continuación, se muestra el Pseudocódigo 1 del método .

Pseudocódigo 1: Pseudocódigo del método en CPU

```
Preprocesamiento e Inicialización;
Mapeo 3D:
for cada punto P de la nube do
    for cada cámra C desde donde es visible el punto P do
        Traslación y rotación del punto P al plano de la cámara C;
        Transformación fisheye del punto P a coordenadas de cámara C;
    end
end
Oclusión:
for cada cámara C do
    for cada punto P de la nube do
        for cada cámara C desde donde es visible el punto P do
            Test de oclusión;
        end
    end
end
Result: Vector de puntos y coordenadas de píxeles
```

Este método plantea una serie de dificultades debido a una irregularidad en la resolución espacial de los datos, problemas inherentes al tratamiento de la oclusión en el modelo 3D y un importante desafío computacional a medida que va creciendo el volumen de los datos adquiridos. Algunos de estos desafíos son tenidos en cuenta para el desarrollo del método en CPU, otros se tratarán de abordar en la solución que se propone en este trabajo basada en GPU. Para tal fin, en primer lugar, es crucial llevar a cabo una descripción detallada del método secuencial que principalmente queda dividido en tres etapas: (1) preprocesamiento de

los datos, (2) mapeo de imágenes sobre la nube de puntos y (3) estudio de la oclusión.

3.2.1 Preprocesamiento

En la fase de preprocesamiento se genera una estructura de datos que almacena el número de puntos que son visibles desde cada cámara. Para ello, se tiene que delimitar el Field of View (FOV) de cada captura que está compuesto por 4 planos. Considerando los parámetros intrínsecos (distancia focal, punto central, centro óptico, etc.) y extrínsecos del sensor multiespectral se define el campo de visión a partir de la posición y orientación de la cámara en la escena (Figura 3.3). Para acelerar la localización de los puntos en la nube, se utiliza un octree.

(a) (b)

Figura 3.3: 3D canvas: (**a**) Cono de visión para una cámara multiespectral; y (**b**)Visualización de cámaras multiespectral y nube de puntos en la escena.

Junto a lo anterior, para realizar el proceso de mapeo es necesario que estableces las cámaras y la nube de puntos estén en el mismo sistema de referencia. No obstante, el alineamiento de ambos conjuntos de datos ya está resuelto en nuestro dataset y no se requiere de ninguna otra transformación geométrica para proceder al mapeo. En la Figura 3.3 se pueden ver tanto la nube de puntos como el conjunto de cámaras que representan cada una de las tomas que fueron realizadas sobre el área de estudio.

3.2.2 Inicialización de estructuras

En esta sección se describen el conjunto de estructuras utilizadas para el cálculo del mapeo y oclusión. La Figura 3.4 ilustra las estructuras más relevantes usadas por el método. En primer lugar, se genera un vector de tamaño igual a la nube de puntos y en cada posición se almacena la posición del punto 3D y su color RGB. En segundo lugar, se crea un vector de

imágenes que guarda información relativa a la posición y orientación de la cámara (parámetros extrínsecos) y los coeficientes de deformación y punto central de la imagen (parámetros intrínsecos). En tercer lugar, se obtiene un vector denominado puntos-cámaras resultante del preprocesamiento de los datos. En esta estructura se tiene una correlación directa entre cada punto y las cámaras desde las que pueden ser vistos al estar dentro de su campo de visión. Por último, la estructura de proyecciones es utilizada para almacenar el resultado del mapeo y la oclusión. En cada posición del vector se almacena el identificador del punto y su correspondiente coordenada de píxel, el identificador de la cámara sobre la que se proyecta, la distancia euclídea entre la posición del punto 3D y la cámara y una variable binaria que es utilizada durante el test de oclusión.

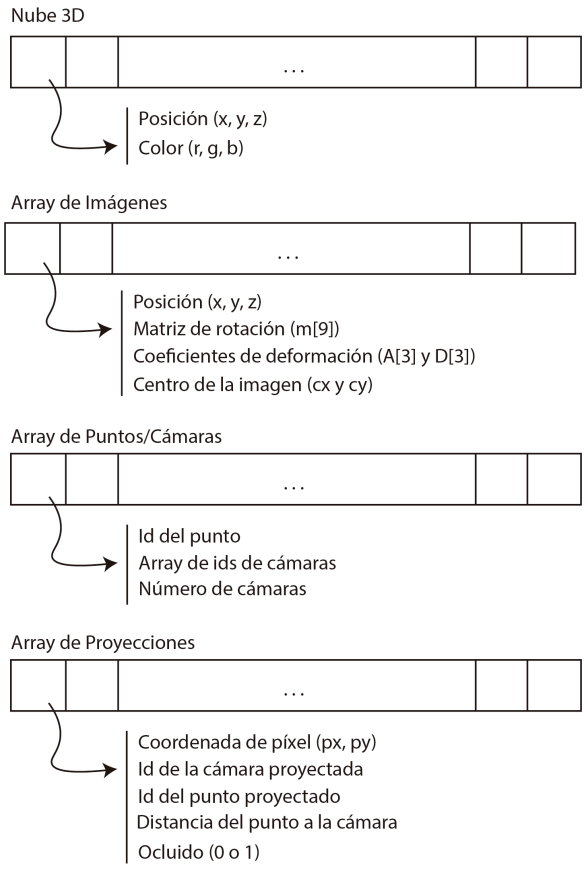

Figura 3.4: Conjunto de estructuras de datos principales utilizadas en el método.

3.2.3 Mapeo 3D

En esta sección se describe la parte del método dirigida al mapeo de imágenes multiespectrales sobre la nube de puntos. En definitiva, este cálculo consiste en: (1) posicionar la cámara en el origen de coordenadas, (2) proyectar la nube de puntos al plano de la imagen aplicando

el modelo de deformación *fisheye*. A continuación, se especifica el siguiente Pseudocódigo 2 descriptivo de esta parte del método.

Pseudocódigo 2: Pseudocódigo del método de mapeo 3D

```
Proyección** proyecciones;
for cada punto P contenido en el vector Punto/Cámara do
    for cada cámara C desde donde es visible el punto P do
        Traslación:
        var pos (x,y,z) //Posición del punto P
        pos -= Posición de la cámara ;
        Rotación:
        pos *= Transpuesta de la matriz de rotación de la cámara.
        Proyección Fisheye:
        Cálculo del polinomio de deformación
        var pol = D[0] + D[1] * theta + D[2] * theta * theta + D[3] * theta * theta *
          theta;
        Transformación de coordenadas:
        var xh = (pol * x) / r;
        var yh = (pol * y) / r;
        var px = A[2] * xh + A[3] * yh + cy;
        var py = A[0] * xh + A[1] * yh + cx;
        punto-proyectado[c] = p ;
    end
    proyecciones[p] = punto-proyectado ;
end
Result: Vector de puntos proyectados
```

En primer lugar, se lleva a cabo la transformación geométrica de la escena para localizar cada cámara en el origen de coordenadas. Para ello, se lleva a cabo una traslación y una rotación sobre el modelo 3D. El vector de traslación es la diferencia entre la posición de la cámara y cada punto de la nube. La matriz de rotación viene definida por los ángulos de inclinación de la cámara en cada uno de los ejes.

En segundo lugar, se calcula la proyección de cada punto a su correspondiente coordenada de imagen. Para ello, hay que tener en cuenta el modelo ojo de pez presente en la imagen. En concreto, la lente del sensor espectral con una focal de 4 mm presenta una deformación en la imagen con el fin de cubrir una mayor extensión de terreno en cada captura. Se parte de la distancia focal (f), que será la distancia entre el centro óptico (c_o) y el punto principal de la imagen (c_p). En la Figura 3.5, se muestra la gráficamente el modelo de deformación.

Atendiendo por tanto a las características particulares de la lente del sensor multiespectral, en el cálculo de la proyección del punto a la imagen se aplica el modelo de deformación fisheye que es parametrizado atendiendo a valores conocidos del sensor. Así, para cada punto si la

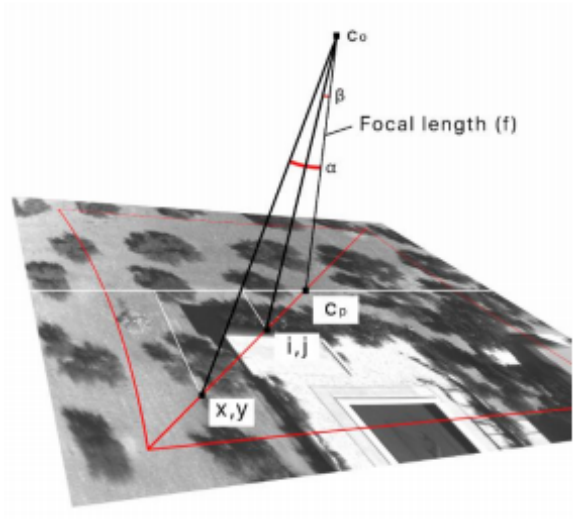

Figura 3.5: Proyección de un punto sobre una imagen con deformación fisheye.

coordenada de imagen resultante está dentro del espacio de la imagen se considera válida. Esta comprobación es hecha para garantizar la inclusión o exclusión de puntos próximos a los bordes de la imagen. Como resultado de este proceso, se genera un fichero de salida en el que se establece una correlación directa entre un punto 3D de la nube y los píxeles de las imágenes desde los que a priori son visibles. En la Figura 3.6 se muestra un ejemplo sobre otro dataset en el que se proyectan todos los puntos visibles desde una de las capturas que fueron tomadas.

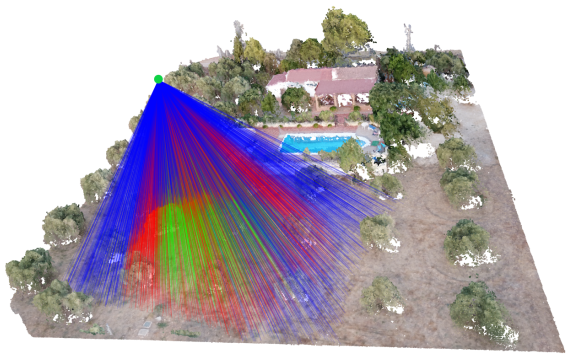

Figura 3.6: Proyección de un conjunto de puntos 3D sobre una cámara.

3.2.4 Oclusión

Una vez desarrollado el método para el mapeo de puntos 3D a coordenadas de imagen, hay que testear si existe oclusión para cada una de las proyecciones. Este test es necesario

implementarlo puesto que hay que cerciorarse que los puntos proyectados no son ocluidos por otros más cercanos a la cámara. Para cada píxel de la imagen se permite que únicamente un punto sea proyectado sobre él. Esta detección de oclusión se basa en el cálculo de la distancia entre el punto y su correspondiente cámara. En cada píxel se implementa un buffer de profundidad de tal forma que al final el punto elegido será aquel que sea más próximo a la posición de la cámara. Un caso típico de oclusión en el conjunto de datos de estudio es que los puntos del terreno sean ocluidos por punto de las copas de los árboles. El test de oclusión que integra el método estudiado se describe mediante el siguiente Pseudocódigo 3.

Pseudocódigo 3: Pseudocódigo del método en CPU

```
var matriz-oclusión[dimensión de la imagen];
for cada cámara k do
    for cada punto P contenido en el vector Punto/Cámara do
        for cada cámara C desde donde es visible el punto P do
            if projections[p][c] = k then
                var dist = projections[p][c].dist ;
                var px =projections[p][c].px ;
                var py =projections[p][c].py ;
                if dist < matriz-oclusión[px * nfilas + py] then
                    proyección no es ocluida;
                    matriz-oclusión[px * nfilas + py] = dist ;
                else
                    proyección es ocluida;
                end
            end
        end
    end
end
```

Result: Vector de puntos proyectados validados por el test de oclusión

3.2.5 Programación OpenMP

Previo a la computación en GPU, en este proyecto se lleva a cabo una paralelización del método en CPU utilizando la API OpenMP. Aunque el objetivo de este trabajo no consistía en avanzar en esta dirección, se consideró de interés hacer una comparativa con una solución sencilla de paralelización en CPU. OpenMP librería es una colección de directivas, rutinas y variables de entorno para programas en Fortran, C y C ++. Actualmente es el estándar utilizado para programación de sistemas de memoria compartida.

El enfoque de paralelización implementado persigue que cada núcleo de la CPU calcule el conjunto de proyecciones asociadas a un punto 3D de la nube. Atendiendo a la estructura

obtenida en la fase de preprocesamiento, por ejemplo, si el punto P1 es visto por 15 cámaras y el P2 por 3 cámaras, el primer núcleo calculará las 15 proyecciones del punto P1 y el segundo núcleo calculará las 3 proyecciones correspondientes a P2. Para ello, se utiliza la sentencia *pragma parallel for* sobre el bucle principal en el que se itera cada punto de la estructura de preprocesamiento. Como variable compartida entre todos los núcleos se define la nube de puntos y la estructura de preprocesamiento, de solo lectura, utilizada para chequear las cámaras asociadas a cada punto 3D.

En cuanto al cálculo de la oclusión, el objetivo es que cada núcleo lleve a cabo el test de oclusión para cada cámara. De este modo, se añade un nuevo *pragma parallel for* sobre el bucle que itera el conjunto de imágenes del dataset. Se considera como variables públicas el vector de proyecciones puesto que cada núcleo accede a posiciones distintas del *array*. La aceleración conseguida siguiendo este enfoque es presentada y comparada con versión secuencial y la paralelización basada en GPU en el Capítulo 5.

3.2.6 Limitaciones del enfoque secuencial

En esta sección se presentan las principales limitaciones encontradas en el método que tienen un efecto negativo sobre el rendimiento final de la aplicación. Fundamentalmente, la principal limitación del método secuencial tiene que ver con el manejo de grandes volúmenes de datos. El cálculo de la proyección de cada punto 3D de la nube y el test de oclusión son operaciones que pueden ser aceleradas significativamente aplicando un alto grado de concurrencia (paralelismo de datos). El método secuencial itera las estructuras, previamente presentadas, con el fin de calcular la coordenada de cada punto 3D en el conjunto de imágenes en las que es visible. De nuevo, estas estructuras son iteradas para el cálculo de la oclusión.

En lo que respecta a la naturaleza de los datos de entrada, tanto el tamaño de la nube de puntos como el número de imágenes puede crecer exponencialmente a medida que se amplíe el área cubierta. En concreto, el número de puntos 3D también está correlacionado con la resolución espacial del sensor que puede verse incrementada si se toman imágenes a menor altura de vuelo.

Por esta razón se explora un enfoque basado en computación paralela que agilice los cálculos implementados por el método. El cálculo de la proyección de cada punto sobre el plano de la imagen no tiene ningún tipo de dependencia por lo que se puede ser paralelizado entre el número de núcleos disponibles. Del mismo modo, el cálculo de la oclusión para cada píxel es independiente del resto por lo que no existe ninguna barrera en el cálculo que imposibilite la computación en paralelo. En consecuencia, se explora una solución basada en GPU que es descrita en el siguiente capítulo.

Capítulo 4

Optimización del método en GPU

En este capítulo se describe la solución que se propone para la computación en paralelo del método de estudio aprovechando la capacidad de cómputo proporcionada por la GPU. Como parte de este desarrollo se han llevado a cabo tres implementaciones distintas y una evaluación final del rendimiento analizando las ventajas y desventajas de cada una de ellas.

4.1 Introducción

4.1.1 GPGPU

La programación de GPU es especialmente adecuada para solucionar problemas que se pueden expresar como cálculos paralelos sobre un conjunto de datos (paralelismo de datos). El objetivo es el uso simultáneo de múltiples recursos para realizar operaciones concurrentes. El enfoque de paralelización que se propone en este trabajo está basado en CUDA (Compute Unified Device Architecture). CUDA es una plataforma de computación en paralelo desarrollada por NVIDIA que incluye un conjunto de herramientas que posibilitan a los programadores codificar algoritmos en GPU. Frente a otras alternativas como OpenACC u OpenCL para la programación paralela en GPUs, CUDA es una de las APIs más ampliamente utilizados en soluciones GPGPU. En el estado del arte existen un amplio número de trabajos que han realizado estudios comparativos en términos de rendimiento entre CUDA y el resto de plataformas/estándares de programación paralela existentes [27]. La elección de CUDA viene fundamentalmente motivada por el interés del autor de este trabajo por profundizar su aprendizaje en esta materia.

Una de las limitaciones a tener en cuenta de CUDA es que se requiere de un hardware específico, en concreto una tarjeta NVIDIA. En paralelo al desarrollo de nuevas tarjetas y arquitecturas de NVIDIA, esta compañía ha ido actualizando las versiones de CUDA con el objetivo de maximizar la aceleración de cálculos complejos y añadir nuevas capacidades de uso. Actualmente, esta API se encuentra en la versión 11, lanzada en mayo del 2020. En este trabajo, se ha utilizado la versión de CUDA 10.2 y la tarjeta NVDIA Titan V que integra la

microarquitectura Volta, lanzada el 7 de diciembre de 2017. parte de la familia Turing (Figura 4.1). Entre las características más relevantes de esta GPU cabe destacar una capacidad de memoria de 8 GB GDDR6 a 14 GHz, 2304 núcleos y un rendimiento de 7.5 TFLOPS.

Figura 4.1: Chip GV100 de la arquitectura Volta

4.1.2 Fundamentos de la Programación Paralela en GPU

En el modelo de programación paralela CUDA la computación de los datos es realizada por una serie de programas ejecutados en la GPU denominados *kernels*. La GPU trabaja como un coprocesador y los datos deben ser transferidos desde la CPU (host) a la GPU (device) (Figura 4.2). El workflow de CUDA viene marcado por los siguientes pasos: (1) reservar memoria en el host y en el device, (2) transferir datos de la memoria de la CPU a la GPU, (3) ejecutar el *kernel* CUDA en la GPU, (4) otras computaciones en paralelo en CPU y (5) transferir los resultados desde la GPU a la CPU.

En lo que respecta a las buenas prácticas recomendadas para la programación paralela basada en GPU, en este trabajo se ha tratado en la medida de lo posible minimizar el número de transferencias entre la CPU y la GPU, tener un acceso coalescente de los datos, y maximizar el uso de memoria compartida en GPU. La memoria global es la memoria de mayor tamaño de la GPU, pero se caracteriza por una mayor latencia. Esta memoria se encuentra fuera del chip de procesamiento y es el medio principal para la transferencia de datos entre el *host* y el *device*. Esta transferencia de datos tiene una penalización importante en cuanto al rendimiento y hay

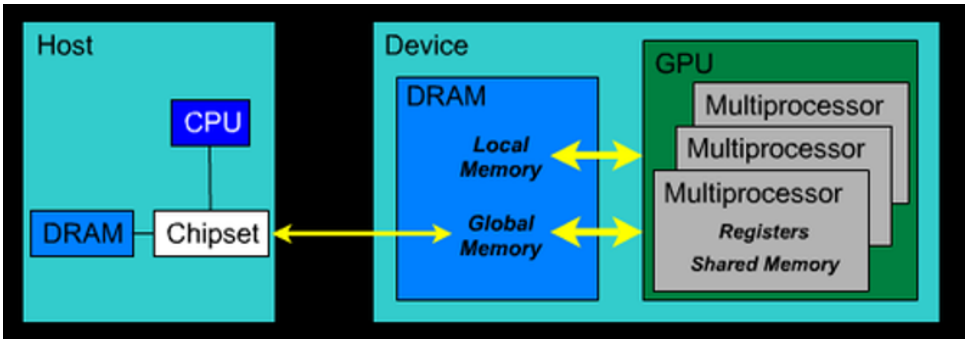

Figura 4.2: CUDA: Modelo de programación.

que tratar de reducirlas en número, así como llevarlas a cabo de forma asíncrona junto a la ejecución del *kernel* en la GPU. En cuanto a los datos almacenados en memoria global de la tarjeta gráfica, estos son visibles por cualquier *thread* de cualquier bloque que se encuentre en ejecución en la GPU. Cuando un *warp* [1] ejecuta una instrucción que accede a la memoria global, si una o más de estas transacciones de memoria dependen del tamaño del bloque leído por cada *thread* y la distribución de las direcciones de memoria se realiza a través de todos los *threads*, se produce un acceso coalescente. Ateniendo a este conjunto de buenas prácticas y conociendo la naturaleza del problema en las siguientes secciones se presenta la paralelización del método que se ha desarrollado en este proyecto.

4.1.3 Motivación

Una vez presentado el contexto de trabajo, centrándonos en las características propias del método de estudio, se considera que el conjunto de cálculos llevados a cabo puede acelerarse significativamente utilizando un enfoque basado en GPU. En el campo de la informática gráfica la tarjeta gráfica se ha utilizado para el renderizado de nube de puntos sobre la que se aplican técnicas de mapeo de texturas para la visualización de modelos 3D haciendo considerando el pipeline de rendering [28, 29]. No obstante, estas soluciones están orientadas fundamentalmente a la visualización fotorrealista de modelos en una escena virtual.

En este trabajo, se propone una solución para la proyección de imágenes y test de oclusión en la escena basada en CUDA puesto que se considera que una solución más versátil con el fin de posibilitar el procesamiento de la información en remoto e incluso repartir la computación de los datos entre múltiples GPUs a medida que crezca el volumen del dataset, tanto nube de puntos como imágenes. Teniendo en cuenta lo anterior, se propone una paralelización del método, así como una nueva arquitectura del sistema que son descritas en la siguientes secciones.

[1] *Warp:* conjunto de hilos (32) donde todos los hilos se ejecutan de forma sincronizada (SIMD)

4.2 Arquitectura del Sistema

Junto a la paralelización del código, se considera necesario plantear una nueva arquitectura del sistema con el fin de proponer una solución más robusta con la capacidad de aprovechar la capacidad de cómputo disponible en remoto. Hasta el momento se llevaba a cabo todo el procesamiento de los datos en la máquina cliente lo que en ocasiones supone una limitación importante en función de las prestaciones hardware disponibles. En este proyecto se ha desarrollado una solución válida para recibir los datos alojados en en servidor de almacenamiento, procesar el conjunto de datos en un cluster o máquina en remoto, y enviar los resultados a los dispositivos cliente en los que se visualizaran tales resultados. La Figura 4.3 ilustra un esquema de la arquitectura propuesta.

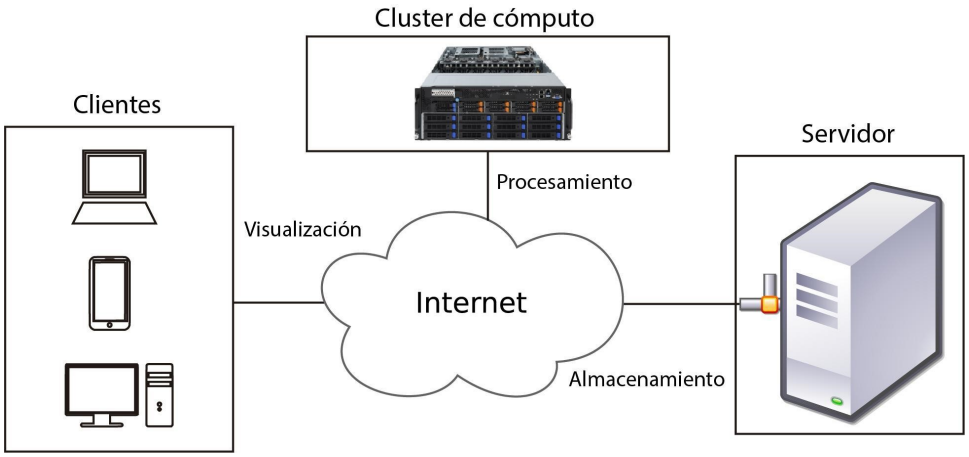

Figura 4.3: Arquitectura del sistema.

4.3 Desarrollo del Método en Paralelo

Durante la fase de codificación del método se han desarrollado tres versiones estables. Inicialmente, se implementó una primera versión (v.01) sobre la que se fueron proponiendo diferentes optimizaciones atendiendo a los análisis de rendimiento y varios cambios para resolver algunos problemas relativos al cálculo de la oclusión en la escena. Tras aplicar estas mejoras, se obtienen la segunda versión (v.02). En ambas versiones, se usó el mismo tamaño del problema, 12 imágenes multiespectrales y una nube 3D formada por 66.374.475 puntos (dataset descrito en la Sección 3.1. Por último, con el fin de proponer una solución out-of-core, es decir, que la capacidad de memoria de la GPU no limite el tamaño de las nubes de puntos a procesar por el método propuesto, se desarrolló la tercera versión del algoritmo (v.03).

4.3.1 Primera versión (v.1)

En la versión inicial se tenía como principal objetivo maximizar la paralelización en cuanto al cómputo de los datos, lo que implica maximizar el número de hilos en paralelo. Esta solución consta de 2 *kernels*, el primer de ellos realizará la función del mapeo 3D y el segundo de ellos llevará a cabo el test de oclusión.

Kernel: Mapeo 3D

La primera tarea del método consistía en la proyección de los puntos 3D de la nube a coordenadas de imagen. Puesto que este cálculo no tiene ningun tipo de dependencia entre puntos vecinos o píxeles adyacentes, se proponer lanzar un hilo para realizar la transformación de coordenadas de cada punto de la nube. En lo que respecta a la dimensión del bloque, se realizaron pruebas con 32, 64 y 128 hilos por bloque. En cuanto a las dimensiones del grid se aplicó la Formula 4.1.

$$DimGrid = (N + dimblock.x - 1)/dimblock.x \tag{4.1}$$

donde N es el número de puntos contenidos en la nube 3D y *dimblock* el tamaño del bloque.

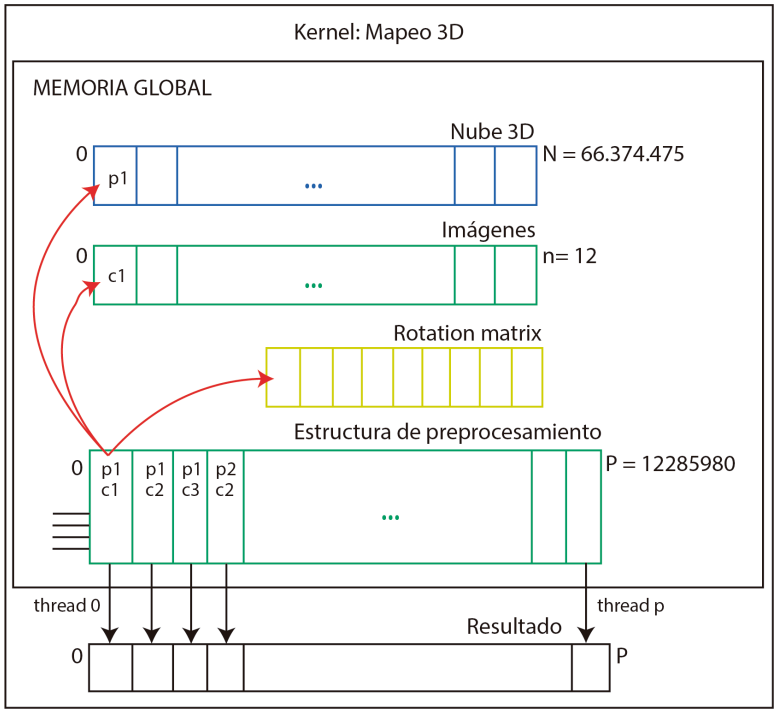

Figura 4.4: Esquema ilustrado sobre la computación en el *kernel* de mapeo 3D

En cuanto a la transferencia de memoria desde la CPU a la GPU este *kernel* utiliza la

estructura generada en la fase de preprocesamiento de los datos (ver Sección 3.3) en la que se tiene una correlación entre cada punto 3D y el conjunto de cámaras desde donde podría ser visible (sin tener en cuenta el test de oclusión). Además, requiere del vector que almacena la nube de puntos, el vector de cámaras y la matriz de rotación de cada cámara. En la Figura 4.4 se ilustra de forma gráfica el cálculo implementado por el *kernel* de mapeo. Como salida del método se obtiene un vector de proyecciones en donde se almacena la coordenada de píxel que tiene cada punto 3D proyectado. En cuanto a esta estructura, resultado del *kernel*, es importante remarcar que un punto puede ser proyectado en más de un píxel de distinta cámara y que puede haber más de un punto proyectado sobre el mismo píxel de una cámara. Estas condiciones han de considerarse a la hora de iterar dicha estructura durante el testeo de la oclusión.

Kernel: Oclusión

El *kernel* de oclusión fue implementado para comprobar si la proyección es válida o no, en el caso de que el punto fuera ocluido por cualquier otro objeto de la escena. Uno de los principales retos al que se tuvo que hacer frente tiene que ver con la iteración de la estructura de salida del *kernel* de mapeo. La oclusión debe realizarse a nivel de píxel, es decir, se deben revisar el conjunto de puntos proyectados para cada píxel de la imagen. El número de puntos que pueden caer en un mismo píxel es irregular por lo que no se puede reservar a priori en la GPU un buffer de profundidad por píxel con una longitud fija. Esta premisa dificultó el proceso de paralelización en torno al cálculo de la oclusión.

Tras estudiar distintas opciones teniendo como objetivo aprovechar la capacidad de cómputo en paralelo de la GPU se tomó la decisión de que cada hilo realizaría la oclusión sobre una linea de la imagen. Para ello, el vector de proyecciones se tuvo que ordenar por filas en CPU, teniendo un registro del número de proyecciones por fila en otro vector secundario. De esta forma, para el conjunto de test, se lanzaron un total de 11.520 hilos en paralelo (imágenes: 12 y filas: 960).

Uno de los punto negativos de este enfoque tiene que ver con el coste de cómputo añadido por la ordenación del vector en CPU, así como un menor alcance en cuanto al nivel de paralelización en comparación con el *kernel* de mapeo. Para tratar de mejorar esta solución se estudiaron una serie de optimizaciones que fueron integradas en la segunda versión del método.

4.3.2 Segunda versión (v.2)

En esta segunda propuesta se mejora el rendimiento de la versión anterior modificando el enfoque de paralelización sobre el test de oclusión y proponiendo una serie de optimizaciones para alcanzar un mejor rendimiento global del método en GPU. En primer lugar, se descarta

el uso de la estructura de preprocesamiento, cuya generación para conjunto de datos más grandes, supondrá un tiempo bastante alto. Por consiguiente, para cada cámara no se conoce el conjunto de puntos del total de la nube que a priori están dentro del cono de visión por lo que el tiempo asociado a dicho proceso es eliminado. Además, se añaden una serie de cambios en el código que supondrán un mejor rendimiento tales como: (1) eliminar la reordenación del vector de proyecciones en CPU, (2) optimización en el manejo de la memoria en CPU, (3) disminuir el tiempo asociado a la transferencia de memoria desde la CPU a la GPU para el cálculo de la oclusión y (4) aceleración del test de oclusión mediante la aplicación de la operación de ordenación y reducción.

Este desarrollo se basa en el uso de dos estructuras siguiendo un de enfoque clave-valor. En primer lugar, se utiliza un vector en el que se va a almacenar un *struct* que tiene como atributo el identificador del punto y la distancia entre el punto y la cámara sobre la que se proyecta. En segundo lugar, se utiliza otra estructura en la que se va almacenar el identificador del píxel para cada imagen sobre el que se ha proyectado el punto. Ambas estructuras tienen un tamaño igual a las dimensiones de la nube de puntos. Como resultado de cada proyección se añade un elemento a estas dos estructuras. Esta operación es realizada por el *kernel* de mapeo. En esta fase de mapeo, se utiliza *UINT-MAX* como marca para aquellos puntos que se proyectan fuera del cono de visión de la cámara que se está procesando, y que por lo tanto no se proyectan sobre ningún píxel.

Una vez ejecutado el *kernel* anterior, y continuando en la GPU, sin ningún tipo de transferencia de datos entre CPU y GPU o cálculo intermedio en la CPU, se implementan dos operaciones para el cálculo de la oclusión: una ordenación de las estructuras y una reducción. Para ello, se utiliza la librería CUB [30], ya integrada en las últimas versiones de CUDA. CUB es una biblioteca de primitivas basada en bloques de hilos concurrentes que ofrece una serie de utilidades para la programación paralela en CUDA. A partir de las operaciones de ordenación y reducción, se pretende realizar el cálculo de la oclusión sobre los píxeles en los que se han proyectado más de un punto, descartando aquellos que son ocluidos. Para ello, en primer lugar, se realiza una ordenación del conjunto de puntos proyectados para localizar aquellos que fueron asignados a un mismo píxel, y que pasen a estar contiguos en memoria, condición necesaria para las operaciones de reducción de CUB. En segundo lugar, se hace una operación de reducción por distancia seleccionando para cada píxel el punto cuya distancia a la cámara sea más pequeña. El resto son automáticamente descartados. Como resultado, se tiene un vector de las dimensiones de la imagen (tamaño = filas x columnas) y sobre cada posición (píxel) se almacena el identificador del punto 3D. En el Pseudocódigo 4.6 se muestra la secuenciación de operaciones más relevantes programadas en esta versión del método.

Una de las principales limitaciones que tiene esta propuesta es la falta de versatilidad para volúmenes de datos muy grandes. Si aumenta el tamaño del conjunto de datos de tal forma

Pseudocódigo 4: Pseudocódigo del método (v.03)

```
Transferir vector de imágenes (metadatos) y matriz de transformación de la CPU a la
  GPU;
for cada cámara do
    Inicialización del vector de valores (id del punto y distancia);
    Inicialización del vector de claves (id del píxel);
    Ejecución del kernel de Mapeo 3D;
    Operación de Ordenación;
    Operación de Reducción;
    Transferir resultado de la GPU a la CPU;
end
Result: Vector de puntos proyectados validados por el test de oclusión
```

que la nube no pueda ser transferida al completo a la GPU por falta de memoria en la tarjeta se requiere de un enfoque *out-of-core*.

4.3.3 Tercera versión (v.3)

En esta última versión se propone una solución en la que se integran el conjunto de optimizaciones anteriores y, además, sea *out-of-core*, es decir, que un aumento en la dimensionalidad de los datos no impida la ejecución del programa en cualquier hardware. El enfoque que se plantea consiste en subdividir la nube de puntos 3D en un conjunto de bloques de tal forma que se paralelicen los cálculos maximizando la ocupación de la memoria global disponible en la tarjeta gráfica. En la Figura 4.5 se muestra el diagrama en el que se desglosa de forma más gráfica el enfoque propuesto.

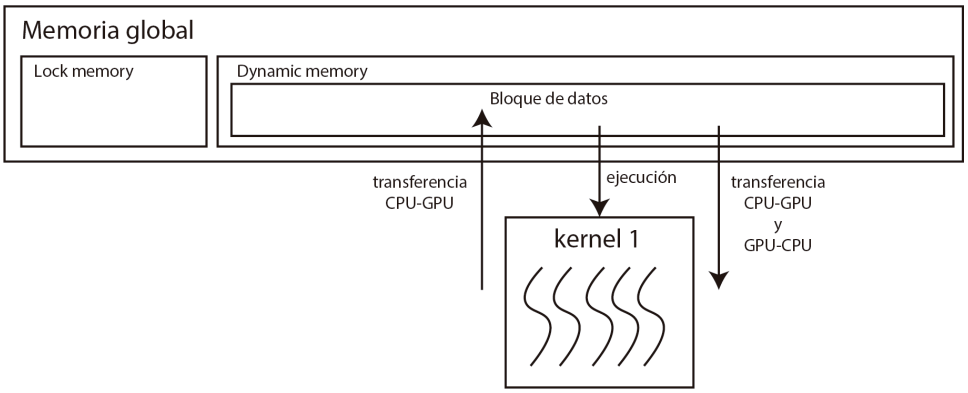

Figura 4.5: Solución *out-of-core* (v.3.1).

El desarrollo de esta solución se desglosa fundamentalmente en dos etapas. En la primera fase se desarrolla el método con el objetivo de maximizar el tamaño de cada subdivisión de la nube ocupando todo el espacio disponible en la GPU. Para ello, la memoria global de la tarjeta

se divide, desde un punto de vista lógico, en dos tipos: *lock-memory* y *dynamic-memory*. Por un lado, como parte de la *lock-memory* se encuentra el conjunto de datos y estructuras que van a permanecer siempre en memoria de la GPU. En cuanto a la *dynamic-memory* se asocian a aquellos datos que van a ser transferidos (CPU-GPU y GPU-CPU) para cada ejecución del *kernel*. Atendiendo a la premisa anterior, la versión 3.1 va considerar el tamaño de cada bloque de datos igual al espacio libre (dynamic-memory) en GPU. De esta forma se consigue una solución *out-of-core* siendo capaces de manejar volúmenes de datos de mayor tamaño a la memoria de la tarjeta gráfica. No obstante, esta versión tiene como principal desventaja el incremento significativo en cuanto al tiempo de transferencia de datos entre el al CPU y la GPU. Esto se debe a que, aunque el número de transferencias puesto que para cada iteración se envía a la GPU un bloque del tamaño igual al espacio de memoria libre en la tarjeta.

Para solventar el problema de rendimiento detectado en la solución anterior, se trabajó en una segunda versión (v.3.2). Este nuevo desarrollo tiene como base la concurrencia asíncrona de las tareas de transferencia de memoria y ejecución de los *kernels* en GPU. En la Figura 4.6 se ilustra el esquema representativo del enfoque propuesto. Las transferencias de memoria ente CPU y GPU serán asíncronas y así, se pretende solapar el tiempo dedicado a la transferencia y al cálculo. Por consiguiente, la memoria dinámica se divide ahora en dos bloques y en cada uno se almacenarán los datos de entrada y salida para la ejecución del primer *kernel* en el *stream 1* y del segundo *kernel* en el *stream 2*. Para obtener el tamaño de cada bloque, se calcula el tamaño de memoria ocupada por las estructuras que estarán durante todo el programa en la GPU (*lock-memory*) y el resto de memoria disponible (*lock-memory*) se divide entre dos partes iguales.

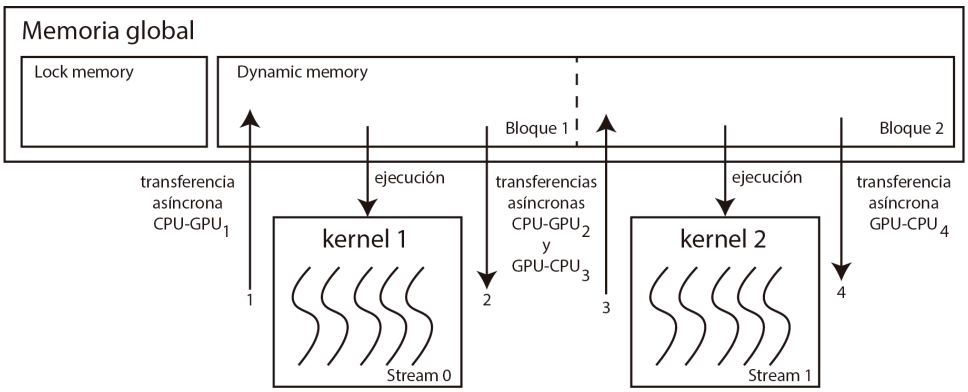

Figura 4.6: Solución *out-of-core* (v.3.2).

Tal y como se indica en el Pseudocódigo 5, se ejecutan n iteraciones, hasta procesar el conjunto de bloques de la nube de puntos. En cada iteración, se procesan pares de bloques, llevándose a cabo las transferencias de memoria y ejecución de los *kernels* de forma asíncrona. Este solapamiento de tareas tiene como principal consecuencia una disminución relevante en

cuanto al tiempo requerido en la versión anterior para la transferencia de los datos.

Pseudocódigo 5: Pseudocódigo del método en CPU

```
for cada iteración do
    Inicialización de cb1;
    Inicialización de cb2;
    cudaMemcpyAsync(cb1,block1, numBytes,cudaMemcpyHostToDevice, StreamA);
    kernel1«<...,StreamA»>();
    cudaMemcpyAsync(cb2,block2, numBytes,cudaMemcpyHostToDevice, StreamB);
    cudaMemcpyAsync(cb2,block2, numBytes,cudaMemcpyDeviceToHost, StreamA);
    kernel1«<...,StreamB»>();
    cudaMemcpyAsync(cb2,block2, numBytes,cudaMemcpyDeviceToHost, StreamB);
end
```

Result: Vector de puntos proyectados validados por el test de oclusión

Capítulo 5

Evaluación del rendimiento

En este capítulo se analiza la aceleración que alcanza la paralelización basada en GPU en comparación con la versión secuencial del método y la paralelización con OpenMP.

5.1 Ejecución secuencial

Una vez descrita cada una de las partes del método propuesto en CPU y desarrollada la paralelización del código con OpenMP se lleva a cabo un análisis del rendimiento utilizando toda la nube de puntos y subconjunto de cámaras, en concreto 12, para agilizar el proceso. Para la ejecución del método se ha utilizado una estación de trabajo (SO: Debian GNU/Linux, Linux Kernel 5.7) que reúne las siguientes características hardware: arquitectura x86-64, 1 procesador Intel con 4 núcleos (cores) i7-4790 CPU @ 3.60GHz (hyperthreading activado con 8 cores virtuales), memoria RAM: 24 GB, Caché L1: 32 KB para datos y 32 KB para instrucciones, Caché L2: 256 KB, Caché L3: 8 MB y GPU: Nvidia TITAN V (Nvidia Driver: 450.57) con 5120 núcleos y VRAM de 12 GB. Esta máquina será utilizada para la ejecución del conjunto de pruebas de rendimiento. La evaluación del rendimiento se ha realizado utilizando dos compiladores distintos: Intel C Compiler (icc^{TM}), en su versión 10.2, y GNU C Compiler (gcc), en su versión 19.1. Ambos posibilitan la compilación del código con varios niveles de optimización descritos en la Tabla 5.1.

Atendiendo a los niveles anteriores, ambos compiladores posibilitan la configuración de todos ellos con algunas peculiaridades que los diferencian. En el caso del compilador de Intel (icc^{TM}) se han añadido los niveles de optimización *-fast* y *-parallel*. En primer lugar, *ifast* compila con *-fast*, que incluye *-Ofast* y *-ipo*. La optimización interprocedural (IPO) pretende reducir o eliminar cálculos duplicados, uso ineficiente de la memoria y simplificar secuencias iterativas como bucles. En segundo lugar, añadiendo la etiqueta *-parallel*, el compilador intenta realizar una paralelización automática del código. Por último, se ha añadido en todos los casos la opción del compilador para generar un código específico para la arquitectura sobre la que

Nivel de optimización	Descripción
-O0	Sin optimización.
-O1	Optimizaciones básicas. Se intenta reducir el tamaño del código y tiempo de ejecución.
-O2	Un nivel mayor de optimización, permitiendo opciones que no involucren una compensación de velocidad o espacio. Nivel común para binarios de producción.
-O3	Se activan opciones que pueden aumentar el tamaño del código. Debido a esto, a veces no es más rápido que -O2.
-Ofast	Se habilita las optimización -ffast-math que no son válidas para todos los programas que cumplen con los estándares.

Tabla 5.1: Niveles de optimización del compilador (los niveles son incrementales: cada uno de ellos incluye las optimizaciones de los niveles anteriores.

se está compilando, en lugar de código genérico x86-64: -march=native en el caso de gcc y -xHost en el caso del compilador de Intel. Esto permite activar opciones de autovectorización en los niveles más agresivas de compilación, con lo que el compilador intenta detectar bucles en el código susceptibles de ser vectorizados, sustituyéndolos por las instrucciones vectoriales correspondientes.

La Tabla 5.2 muestra los tiempos obtenidos diferenciando entre (1) tiempo global, (2) tiempo requerido para el mapeo y (3 tiempo requerido para la oclusión.

	icc			**gcc**		
Nivel de optimización	**Mapeo 3D**	**Oclusión**	**Total**	**Mapeo 3D**	**Oclusión**	**Total**
No opt. (-O0)	4338.51	1548.48	5887.01	4183.01	1303.436	5486.470
Opt level 1 (-O1)	2398.85	565.347	2964.22	1258.02	682.838	1940.879
Opt level 2 (-O2)	903.77	567.345	1471.14	1183.72	674.889	1858.628
Opt level 3 (-O3)	920.96	575.93	1496.92	1019.72	675.301	1695.042
Opt level fast (-Ofast)	873.87	566.84	1440.75	848.16	675.040	1523.220
-ifast	856.70	572.57	1429.30	-	-	-
-parallel	843.93	563.47	1407.43	-	-	-

Tabla 5.2: Medición de tiempos de cómputo (ms) con el compilador gcc y icc^{TM}.

Como cabe de esperar, la compilación sin optimización conlleva un mayor tiempo de ejecución de los binarios generados. En concreto, apenas hay diferencia en cuanto a los tiempos generados por los binarios por ambos compiladores alcanzando un tiempo global entorno a los 5800 milisegundos (ms). Se aprecia una importante mejora de rendimiento en el nivel de optimización 1 (O1) sobre todo en los binarios de gcc alcanzando un tiempo global a los dos segundos. A medida que se van aplicando los niveles de optimización más altos, se observa

que el compilador de Intel logra un mejor rendimiento llegando a reducir el tiempo global de ejecución a 1.42 segundos, ligeramente inferior a los 1.5 segundos conseguidos por el compilador gcc. Considerando el mejor tiempo alcanzado por el compilador icc^{TM}, sin aplicar aún nada de paralelización, se consigue una aceleración de un 75.86% con respecto a la ejecución de los binarios sin optimización aplicada. Por último, en cuanto al intento de paralelización por el compilador de Intel, utilizando la directiva *-parallel*, no se consigue una reducción significativa del tiempo de cómputo global que baja tan solo unas centésimas de segundo, a 1.4 segundos. Tomando estos tiempos como referencia, en las siguientes secciones se muestran los resultados obtenidos tras aplicar los distintos enfoques de paralelización desarrollados.

5.2 Paralelización en CPU

Tras evaluar el rendimiento del método en secuencial aplicando los distintos niveles de optimización posibles para los compiladores gcc e icc ^{TM} se presenta la aceleración conseguida mediante la paralelización en CPU utilizando las directivas OpenMP. Puesto que el procesador utilizado para las pruebas de rendimiento tiene un total de ocho núcleos, se han realizado tres ejecuciones utilizando dos, cuatro y ocho núcleos en paralelo. En la Tabla 5.3 se muestran los tiempos obtenidos.

	icc			**gcc**		
Nº de núcleos	**Mapeo 3D**	**Oclusión**	**Total**	**Mapeo 3D**	**Oclusión**	**Total**
2	759.53	434.29	1193.86	700.24	633.98	1334.25
4	767.38	420.18	1187.6	694.88	587.73	1282.65
8	792.32	391.21	1183.56	535.99	458.45	994.47

Tabla 5.3: Medición de tiempos de cómputo (ms) con OpenMP.

Atendiendo a los tiempos de ejecución, a medida que se aumenta el número de hilos se consigue un menor tiempo para el cálculo del mapeo y el test de oclusión. Si se comparan los resultados entre ambos compiladores, la ejecución de los binarios compilados con gcc alcanzan ligeramente un mejor rendimiento consiguiendo el tiempo más bajo de 994.47 ms. Este tiempo significa un 30.4 % de aceleración con respecto a la ejecución secuencial. Para la ejecución con dos y cuatro hilos esta aceleración se reduce a un 6.6% y un 10.2 % respectivamente.

En la Figura 5.1 se muestra una gráfica en la que se representa la aceleración para la operación de mapeo, el test de oclusión y el tiempo global para la ejecución de dos, cuatro y ocho núcleos de la CPU en paralelo. En cuanto a estos resultados cabe destacar que se consigue una aceleración más significativa sobre la operación de mapeo con ocho núcleos (38.6%) aunque el test de oclusión se acelera en un 19%.

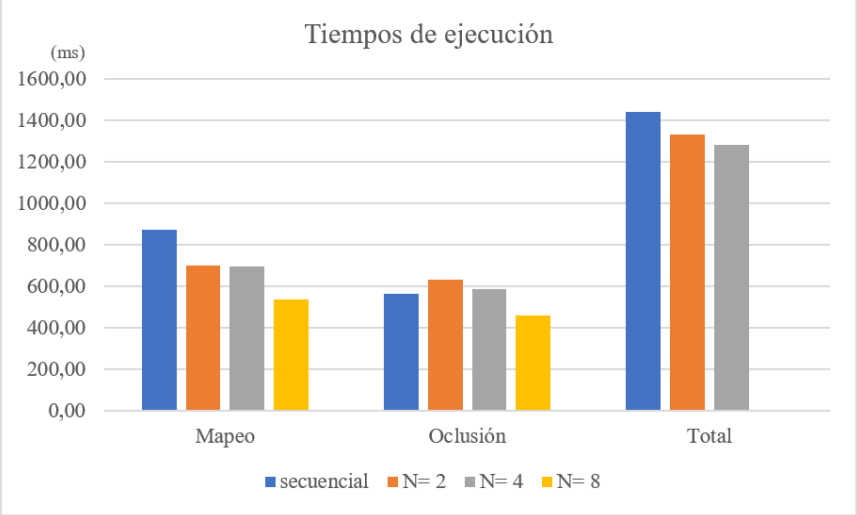

Figura 5.1: Comparativa de rendimiento para cada operación (mapeo, oclusión y total en secuencial y paralelización en CPU (OpenMP.

5.3 Paralelización en GPU

En esta sección se presentan los resultados obtenidos a partir de los desarrollos llevados a cabo en GPU. Para situar al lector, la primera versión del método acelerado por GPU se basaba en la estructura de preprocesamiento y el principal problema estaba relacionado con un tiempo de computación excesivo para el cálculo de la oclusión, en torno a 14 segundos. En cuanto al tiempo para el cálculo del mapeo (incluyendo el tiempo de la transferencia de memoria es de 400.59 ms lo que suponía una aceleración importante con respecto al tiempo en secuencial por lo que esta parte se reutilizó para el desarrollo de la siguiente versión.

Para superar la limitación del primer enfoque se llevó a cabo la segunda versión sobre la que ya se obtiene un rendimiento aceptable tanto para la operación de mapeo como para la operación de oclusión. Tomando como base esta implementación, se desarrolló la versión *out-of-core* (v 3.1 con la que se conseguía subdividir el dataset en varios bloques. Para ver como escala el rendimiento con respecto al número de hilos por bloque se lanzaron tres ejecuciones definiendo el tamaño del bloque a 32, 64 y 128 hilos. En la Tabla 5.4 se muestran los tiempos de cada versión desglosados para cada operación de cálculo y transferencia de memoria.

En cuanto a los tiempos medidos para la versión 2.0 se consigue reducir el tiempo de ejecución a medida que se incrementan el número hilos por bloque consiguiendo el menor tiempo global de 980.18 ms con 128 hilos en paralelo. Si se pone el foco en los resultados relativos a la versión 3.1 se aprecia que esta solución consigue peores tiempos que la anterior fundamentalmente en lo que a la transferencia de memoria. Puesto que nuestro conjunto de

	Mapeo 3D		Oclusión		Transferencia de memoria		Totales	
Hilos/Versión	**v.2**	**v.3.1**	**v.2**	**v.3.1**	**v.2**	**v.3.1**	**v.2**	**v.3.1**
32 hilos	90.37	95.01	352.52	418.94	535.73	4909.89	1366.48	5523.12
64 hilos	90.36	94.75	352.3	420.59	981.56	4582.51	981.06	5203.4
128 hilos	90.38	95.17	352.37	420.73	534.45	4554.2	980.186	5173.82

Tabla 5.4: Medición de tiempos de cómputo (ms comparando las versiones v.2 y v.3.1 (out-of-core síncrona.

datos de entrenamiento cabe al completo en la memoria de la tarjeta gráfica utilizada, para este estudio del rendimiento, se ha limitado manualmente la memoria de la tarjeta a 1GB con el fin de testear el funcionamiento del método *out-of-core*. Como se aprecia en la Tabla 5.4, las transferencias de memoria son las operaciones que presentan un impacto negativo relevante sobre el rendimiento final del método. En concreto estas representan en torno al 85% del tiempo global.

Una vez detectada la operación que resultaba más ineficiente de nuestro enfoque, se desarrolla la última versión de este trabajo (v.3.2 que implementa de forma asíncrona las operaciones de transferencia de memoria y computación de los datos en GPU. Este enfoque consigue solventar el problema de la versión anterior consiguiendo solapar ambas tareas. En la Tabla 5.5 se muestran los resultados de la versión 2.0 (no out-of-core) y la versión 3.2 en términos de tiempos globales de ejecución.

	Tiempo total	
Hilos/Versión	**v.2.0**	**v.3.2**
32 hilos	1366.48	1243.62
64 hilos	981.06	902.717
128 hilos	980.186	893.94

Tabla 5.5: Tiempo global de ejecución de la versión v.2 y v.3.2 en ms

Como se aprecia en la tabla anterior, la versión optimizada *out-of-core* consigue unos resultados bastante óptimos si comparamos con los tiempos de rendimiento que han sido medidos hasta el momento. Para un tamaño de bloque de 128 hilos se alcanza un tiempo de 893.94 ms. Finalmente, si comparamos este tiempo de ejecución con respecto al método secuencial se logra una aceleración de 38 %, un 8% mayor que la solución basada en OpenMP. En la Figura 5.2 se muestra una representación conjunta de los mejores tiempos resultante de la ejecución secuencial, la paralelización con OpenMP y la paralización basada en CUDA.

Finalmente, una vez concluidas todas las pruebas para testear el rendimiento de los desarrollos realizados en este proyecto, se ejecuta el método propuesto en su versión 3.2 sobre el conjunto de datos al completo, es decir, la proyección de 180 imágenes sobre la nube de pun-

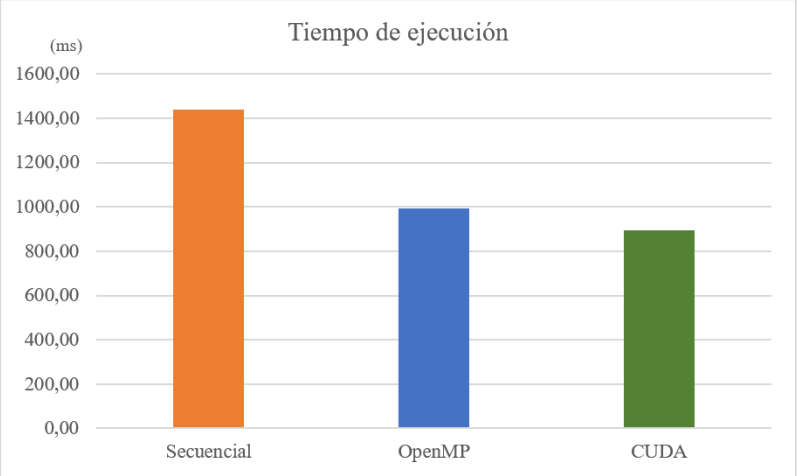

Figura 5.2: Comparativa de tiempos globales sobre el conjunto de entrenamiento: (1) secuencial, (2) OpenMP y (3) CUDA.

tos. En la Figura 5.3, se muestra una comparación entre el tiempo de ejecución del método basado en GPU, el método secuencial y la paralelización con OpenMP.

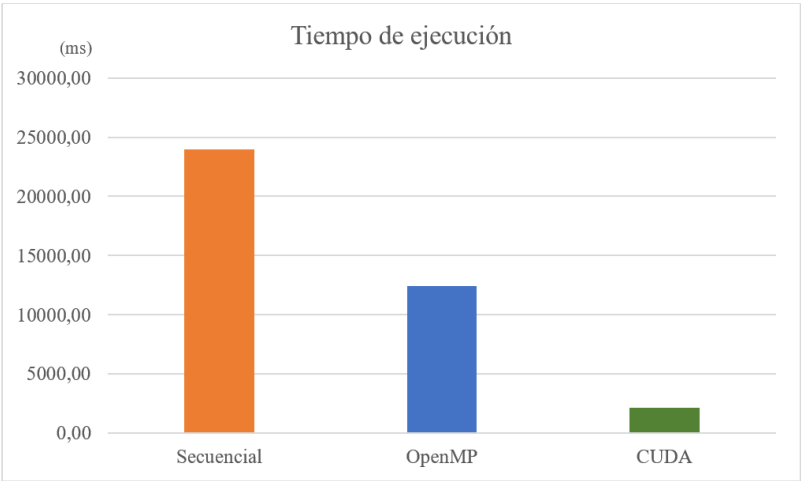

Figura 5.3: Comparativa de tiempos globales sobre *dataset* completo: (1) secuencial, (2) OpenMP y (3) CUDA.

A medida que aumenta la dimensionalidad del conjunto de datos el desempeño de nuestro método desarrollado en CUDA consigue una aceleración de un 91 %, que es bastante mayor a la aceleración alcanzada con OpenMP (ocho hilos en paralelo) que alcanza tan solo un 48%. En términos de tiempos totales, la ejecución secuencial requiere de 23 segundos, la solución basdad en OpenMP alcanza un tiempo de 12 segundos y por último el método propuesto acelerado en GPU logra un tiempo de 2 segundos.

Capítulo 6

Conclusiones

El desarrollo de nuevos métodos con la capacidad procesar grandes volúmenes de datos es hoy día una de las lineas de investigación e innovación que mayor interés despierta en la comunidad científica. La producción de multitud de sensores y cámaras de alta resolución capaces de adquirir información del mundo real con un gran nivel de detalle conlleva la generación de enormes *datasets* que requieren del uso de software específico para procesarlos y obtener los resultados esperados. En este sentido, la programación paralela basada en GPU abre un abanico de posibilidades para abordar un gran número de problemas que son costosos computacionalmente y pueden ser acelerados por GPU. En este marco de trabajo surge el presente proyecto que tiene como principal finalidad el desarrollo de un método para el mapeo de imágenes multiespectrales sobre nubes de puntos de gran tamaño en GPU.

A lo largo de este trabajo se han llevado a cabo varios desarrollos con el fin de obtener una paralelización eficiente del método tanto en el cálculo del mapeo como en el test de oclusión. Como resultado de varias iteraciones de las que se derivan distintas versiones del método, se propone una solución *out-of-core*, es decir, el método se adapta a las especificaciones hardware del equipo utilizado para la computación en paralelo del conjunto de datos. El método propuesto ha sido testeado utilizando un conjunto de pruebas compuesto una nube de 66.374.475 puntos y 12 imágenes. Además, se ha probado una paralelización sencilla en CPU basada en OpenMP para comparar el rendimiento de la paralelización en CPU y GPU. Sobre este conjunto de entrenamiento se obtuvo una aceleración de un 30% en CPU y una aceleración de 38% en GPU. Finalmente, se utilizó el *dataset* completo para ver el comportamiento del método sobre una dimensión real de los datos. Los resultados obtenidos demuestran que la solución que se propone en este proyecto alcanza una aceleración del 91 % significativamente mayor que la aceleración alcanzada con OpenMP (48%) sobre la ejecución secuencial.

Como trabajo futuro se considera de gran interés para este estudio el avance hacia un enfoque multi-gpu con el fin de paralelizar uno o varios conjuntos de datos de forma simultánea en múltiples GPUs. Se pretende así estudiar cómo se comporta el método desarrollado *out-of-*

core sobre varias tarjetas conectadas entre sí por NVLINK y poder así lograr un procesamiento masivo eficiente de datos multitemporales en un cluster de altas prestaciones computacionales.

Apéndices

Apéndice A

Glosario de acrónimos

HPC *High Performance Computing.*

FoV *Fiel of View*

gcc *GNU Compiler Collection*

icc *Intel C Compiler*

CUDA *Compute Unified Device Architecture*

SfM *Structure from Motion*

LiDAR *Light Detection and Ranging o Laser Imaging Detection and Ranging*

API *Application Programming Interface*

Apéndice B

Glosario de terminos

Campo de visión El campo de visión o campo de perspectiva, también conocido por sus siglas en inglés equivalente FOV es la extensión de mundo observable en un momento dado.

Nube de puntos Conjunto de vértices en un sistema de coordenadas tridimensional. Estos vértices se identifican como coordenadas X, Y, y Z y son representaciones de la superficie externa de un objeto.

Mapeo 3D Proceso de correlación entre dos conjuntos de datos en un escenario tridimensional.

Fotogrametría Técnica cuyo objetivo es estudiar y definir con precisión la forma, dimensiones y posición en el espacio de un objeto cualquiera a partir de múltiples imágenes superpuestas entre sí de tal objeto.

Distorsión Fisheye Distorsión visual de la imagen originada por lentes súper gran angular (ángulos de visión muy grandes y distancias focales pequeñas).

Imagen multiespectral Imagen que captura luz de un determinado rango de longitudes de onda del espectro electromagnético, incluidas aquellas no visibles por el ojo humano.

GPGPU Concepto ampliamente utilizado en ciencia de la computación referido al estudio y explotación de las capacidades de cómputo de una tarjeta gráfica.

Aceleración Ratio que relaciona el tiempo de ejecución de un proceso antes y después de una optimización aplicada sobre dicho proceso.

Bibliografía

[1] J. M. Jurado, L. Ortega, J. J. Cubillas, and F. R. Feito, "Multispectral mapping on 3d models and multi-temporal monitoring for individual characterization of olive trees," *Remote Sensing*, vol. 12, no. 7, p. 1106, Mar 2020.

[2] H. Yao, R. Qin, and X. Chen, "Unmanned aerial vehicle for remote sensing applications—a review," *Remote Sensing*, vol. 11, no. 12, p. 1443, 2019.

[3] L. Pádua, J. Vanko, J. Hruška, T. Adão, J. J. Sousa, E. Peres, and R. Morais, "UAS, sensors, and data processing in agroforestry: A review towards practical applications," *International journal of remote sensing*, vol. 38, no. 8-10, pp. 2349–2391, 2017.

[4] H. L. Wiggins, C. R. Nelson, A. J. Larson, and H. D. Safford, "Using lidar to develop high-resolution reference models of forest structure and spatial pattern," *Forest ecology and management*, vol. 434, pp. 318–330, 2019.

[5] J. L. Schonberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4104–4113.

[6] N. Guimarães, L. Pádua, P. Marques, N. Silva, E. Peres, and J. J. Sousa, "Forestry remote sensing from unmanned aerial vehicles: A review focusing on the data, processing and potentialities," *Remote Sensing*, vol. 12, no. 6, p. 1046, 2020.

[7] J. M. Jurado, J. L. Cárdenas, C. J. Ogayar, L. Ortega, and F. R. Feito, "Semantic segmentation of natural materials on a point cloud using spatial and multispectral features," *Sensors*, vol. 20, no. 8, p. 2244, Apr 2020.

[8] J. Jurado, M. Ramos, C. Enríquez, and F. Feito, "The impact of canopy reflectance on the 3d structure of individual trees in a mediterranean forest," *Remote Sensing*, vol. 12, no. 9, p. 1430, May 2020.

[9] G. A. Blackburn, "Hyperspectral remote sensing of plant pigments," *Journal of experimental botany*, vol. 58, no. 4, pp. 855–867, 2007.

[10] S. Pirk, M. Jarząbek, T. Hädrich, D. L. Michels, and W. Palubicki, "Interactive wood combustion for botanical tree models," *ACM Trans. Graph.*, vol. 36, no. 6, pp. 197:1–197:12, Nov. 2017.

[11] Z. Xiaoliang, Z. Guihua, L. Jonathan, Y. Yuanxi, and F. Yong, "3d land cover classification based on multispectral lidar point clouds." *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences*, vol. 41, 2016.

[12] W. Dai, B. Yang, Z. Dong, and A. Shaker, "A new method for 3d individual tree extraction using multispectral airborne lidar point clouds," *ISPRS journal of photogrammetry and remote sensing*, vol. 144, pp. 400–411, 2018.

[13] X. Shen, L. Cao, B. Yang, Z. Xu, and G. Wang, "Estimation of forest structural attributes using spectral indices and point clouds from uas-based multispectral and rgb imageries," *Remote Sensing*, vol. 11, no. 7, p. 800, 2019.

[14] Y. Tian, W. Song, L. Chen, Y. Sung, J. Kwak, and S. Sun, "A fast spatial clustering method for sparse lidar point clouds using gpu programming," *Sensors*, vol. 20, no. 8, p. 2309, Apr 2020.

[15] X. Hu, X. Li, and Y. Zhang, "Fast filtering of lidar point cloud in urban areas based on scan line segmentation and gpu acceleration," *IEEE Geoscience and Remote Sensing Letters*, vol. 10, no. 2, pp. 308–312, 2013.

[16] J. S. Mueller-Roemer, "Gpu data structures and code generation for modeling, simulation, and visualization," Ph.D. dissertation, 2019.

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[18] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[19] J. A. Fraire, A. Ferreyra, and C. Marques, "Opencl overview, implementation, and performance comparison," *IEEE Latin America Transactions*, vol. 11, no. 1, pp. 274–280, 2013.

[20] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017.

[21] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581*, 2010.

[22] A. J. Rueda, J. M. Noguera, and A. Luque, "A comparison of native gpu computing versus openacc for implementing flow-routing algorithms in hydrological applications," *Computers & Geosciences*, vol. 87, pp. 91–100, 2016.

[23] Y. Yuan, X. Yang, W. Wu, H. Li, Y. Liu, and K. Liu, "A fast single-image super-resolution method implemented with cuda," *Journal of Real-Time Image Processing*, vol. 16, no. 1, pp. 81–97, 2019.

[24] S. Aydin, R. Samet, and O. F. Bay, "Real-time parallel image processing applications on multicore cpus with openmp and gpgpu with cuda," *The Journal of Supercomputing*, vol. 74, no. 6, pp. 2255–2275, 2018.

[25] I. Sa, M. Popović, R. Khanna, Z. Chen, P. Lottes, F. Liebisch, J. Nieto, C. Stachniss, A. Walter, and R. Siegwart, "Weedmap: A large-scale semantic weed mapping framework using aerial multispectral imaging and deep neural network for precision farming," *Remote Sensing*, vol. 10, no. 9, p. 1423, Sep 2018.

[26] D. Parrot, "Parrot sequoia," Retrieved 12/04/2019, 2019, from https://www. parrot. com/business ..., Tech. Rep., 2019.

[27] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 2017, pp. 1–6.

[28] H. Shum and S. B. Kang, "Review of image-based rendering techniques," in *Visual Communications and Image Processing 2000*, vol. 4067. International Society for Optics and Photonics, 2000, pp. 2–13.

[29] P. Rosenthal and L. Linsen, "Image-space point cloud rendering," in *Proceedings of Computer Graphics International*, 2008, pp. 136–143.

[30] D. Merrill, "Cub v1. 5.3: Cuda unbound, a library of warp-wide, blockwide, and device-wide gpu parallel primitives," *NVIDIA Research*, 2015.