Departamento de electrónica y computación
Escuela Técnica Superior de Ingeniería

TRABAJO FIN DE MÁSTER
MÁSTER INTERUNIVERSITARIO EN
COMPUTACIÓN DE ALTAS PRESTACIONES

Convoluciones 3D en paralelo

Estudiante: Alberto Fernández López
Director/a/es/as: José Carlos Cabaleiro Domínguez
Alberto Manuel Esmorís Pena

Santiago de Compostela, 26 de mayo de 2024.

A mi esposa e hijos, cuyo apoyo y comprensión han sido el impulso detrás de cada paso en este viaje académico. Este logro también les pertenece.

Agradecimientos

- Agradecimientos al Centro de Supercomputación de Galicia por ofrecer la infraestructura de computación utilizada en este trabajo.
- Agradecimientos a la School of Engineering de la University of Dayton por ofrecer el conjunto de datos utilizado para la experimentación realizada en este trabajo.
- Agradecimientos a mis directores de proyecto por su dedicación, orientación y toda la sabiduría compartida, aspectos que han sido fundamentales para el desarrollo de este trabajo.

Resumen

En este Trabajo de Fin de Máster se construye y optimiza un algoritmo de convolución 3D que se encarga de procesar nubes de puntos LiDAR aéreo que permita identificar objetos similares según el patrón de entrada dado.

Concretamente, se utiliza un conjunto de datos correspondiente a zonas residenciales y, para evaluar el funcionamiento del algoritmo, se marca como objetivo la identificación de edificaciones.

Tras la construcción del algoritmo de convolución y el ensayo con diferentes estructuras de representación de la nube de puntos, se aplican optimizaciones de paralelización en memoria compartida tratando de mantener un consumo de memoria reducido.

Tras aplicar las técnicas consideradas, se consigue reducir el tiempo de ejecución de la convolución en un 98%, utilizando para ello 64 núcleos en 2 procesadores con OpenMP.

Abstract

In this Master Thesis, a 3D convolution algorithm is built and optimized to process aerial LiDAR point clouds in order to identify similar objects according to the given input pattern.

Specifically, a dataset corresponding to residential areas is used and, in order to evaluate the performance of the algorithm, the identification of buildings is set as a target.

Following the construction of the convolution algorithm and experimentation with different point cloud representation structures, optimizations for shared memory parallelization are applied trying to achieve a low memory consumption.

After applying the considered techniques, the execution time of the optimized part is reduced by 98%, using 64 cores in 2 processors with OpenMP.

Palabras clave:

- Convolución 3D
- Nube de puntos
- Reconocimiento de objetos
- LiDAR
- Computación paralela
- OpenMP
- HPC

Keywords:

- 3D Convolution
- Point cloud
- Object recognition
- LiDAR
- Parallel computing
- OpenMP
- HPC

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Contexto

En el ámbito de la computación gráfica y el procesamiento de imágenes, el cálculo de convoluciones en nubes de puntos 3D es una tarea fundamental con múltiples aplicaciones, muchas de las cuales están centradas en el reconocimiento de objetos o estructuras. Entre estas aplicaciones se encuentran:

- Percepción del entorno (robótica y vehículos autónomos)
- Interacción humano-computadora (realidad virtual y aumentada)
- Análisis de datos 3D (medicina y biología)
- Procesamiento de señales 3D (audio y vídeo)
- Inspección y control de calidad (industria y seguridad)

Una **nube de puntos** es un conjunto de datos que representan la posición de puntos en el espacio tridimensional. Estos puntos pueden ser obtenidos mediante diversas técnicas de escaneo, como la fotogrametría, el escaneo láser o el escaneo por luz estructurada. Cada punto en la nube de puntos tiene coordenadas (x, y, z) que indican su posición en el espacio tridimensional, y a menudo también se pueden asociar otros atributos, como intensidad de color, reflectividad o información de textura. [1]

El término ”LiDAR” (acrónimo del inglés *Light Detection and Ranging* o *Laser Imaging Detection and Ranging*) [2] se refiere a un método utilizado para medir distancias mediante el uso de pulsos láser. El LiDAR se emplea para escanear el entorno y generar una nube de puntos tridimensional que representa la superficie de los objetos detectados. La alta precisión y resolución del LiDAR lo hacen especialmente útil para la generación de modelos digitales del terreno y la creación de mapas en 3D de alta fidelidad. [3]

El conjunto de datos *Dayton Annotated Laser Earth Scan* (DALES), es un conjunto de datos LiDAR aéreos a gran escala con casi medio billón de puntos que abarcan un área de 10 kilómetros cuadrados, y que ha sido creado específicamente para aplicaciones de investigación. Contiene cuarenta escenas de datos aéreos densos y etiquetados que abarcan múltiples tipos de escenas, incluidas las urbanas, suburbanas, rurales y comerciales. Los datos fueron etiquetados a mano por un equipo de técnicos expertos en LiDAR en ocho categorías: suelo, vegetación, coches, camiones, postes, líneas eléctricas, vallas y edificios. [4]

En la figura 1.1 se muestra la escena que se utilizará como base de pruebas en este trabajo. Los colores observados corresponden a la clasificación de los objetos que contiene.

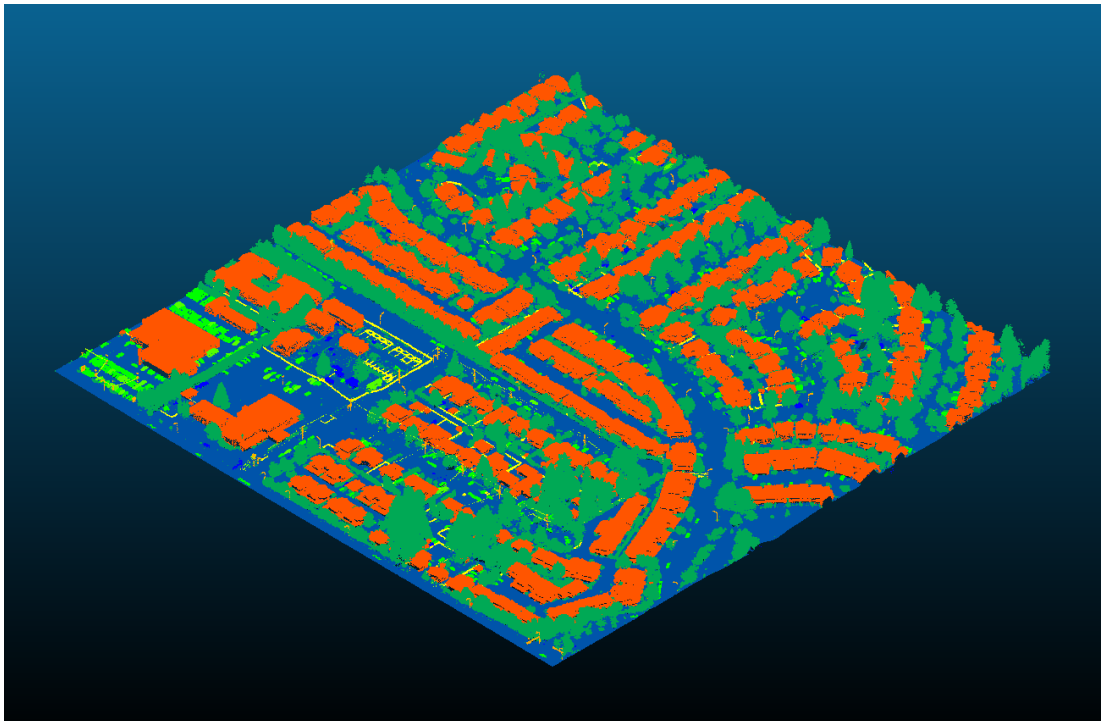


Figura 1.1: Vista clasificada de la escena 5080_54400 del conjunto de datos DALES

El procesamiento de nubes de puntos 3D suele ser computacionalmente intensivo debido a la gran cantidad de datos involucrados y la complejidad de las operaciones requeridas. Esto incluye tareas como filtrado de ruido, alineación de múltiples escaneos, segmentación de objetos, registro de puntos, extracción de características y, como ya se ha introducido, el cálculo de convoluciones.

En este contexto, este trabajo aborda el desarrollo de un programa destinado a calcular convoluciones en nubes de puntos 3D para el reconocimiento de objetos y, como objetivo principal, su paralelización para mejorar la eficiencia del proceso sin pérdida de calidad.

1.2 Trabajos relacionados

En esta sección se revisan algunos de los principales trabajos relacionados con el cálculo de convoluciones en nubes de puntos 3D, tratando de identificar aquellos enfoques relevantes que han contribuido al avance en el procesamiento y análisis de estos datos tridimensionales. Estos trabajos proporcionan una base sólida para el desarrollo de métodos eficientes y precisos para el procesamiento de nubes de puntos 3D. Antes de presentarlos, conviene aclarar que en este trabajo se desarrolla un método propio de procesamiento directo, que utiliza vóxeles para una representación eficiente de la nube de puntos, que ha sido diseñado desde cero sin depender de librerías o implementaciones previas, y que pretende mejorar la eficiencia del cálculo de convoluciones mediante técnicas de paralelización.

La vasta mayoría de los trabajos previos examinados se basan en redes neuronales convolucionales (CNNs), un tipo de red neuronal artificial (ANN) especialmente diseñada para el procesamiento de imágenes. Las CNNs son capaces de extraer características y patrones complejos a partir de imágenes digitales, lo que las convierte en herramientas poderosas para una amplia gama de tareas relacionadas con la visión artificial. Aunque no es el enfoque de este trabajo, en el que se propone un procesamiento directo, resulta imposible aproximarse al estado del arte sin mencionar esta tecnología.

En cuanto a la representación de datos, el uso de representaciones volumétricas es una aproximación popular en el procesamiento de nubes de puntos. *VoxNet* [5] es una red neuronal convolucional (CNN) que convierte nubes de puntos en vóxeles para realizar reconocimiento de objetos en tiempo real. Por su parte, *SegCloud* [6] combina representaciones voxelizadas con convoluciones 3D para la segmentación semántica, demostrando eficacia en escenarios de alta complejidad. *3D ShapeNets* [7] utiliza CNNs sobre representaciones volumétricas para la clasificación de objetos 3D. *PointCNN* [8] introduce una arquitectura que aprende transformaciones de puntos para aplicar convoluciones estándar, mejorando la eficiencia y precisión en la clasificación de nubes de puntos.

El uso de octrees también ha sido explorado para mejorar la eficiencia del procesamiento volumétrico. *OctNet* [9] utiliza una estructura de datos basada en octrees que permite la representación jerárquica de las nubes de puntos. OctNet mejora la eficiencia del almacenamiento y el cálculo al dividir el espacio 3D en una jerarquía de cubos adaptativos, aplicando convoluciones en estos cubos de manera eficiente.

Otro enfoque relevante es el uso de métodos basados en grafos. *Dynamic Graph CNN (DGCNN)* [10] construye grafos dinámicos de vecinos más cercanos y aplica convoluciones en los bordes del grafo, capturando de manera efectiva las relaciones locales entre los puntos. Este enfoque ha demostrado ser altamente eficaz para la segmentación semántica y la clasificación de nubes de puntos.

Por otro lado, se han desarrollado arquitecturas que procesan directamente las nubes de puntos sin necesidad de conversiones a estructura intermedias. *PointNet* [11] introduce una arquitectura pionera que procesa directamente nubes de puntos utilizando una red neuronal que opera en puntos individuales combinando características globales y locales. Con *PointNet++* [12] se extendió este trabajo, incorporando jerarquías de características locales para mejorar el procesamiento de nubes de puntos con estructuras más complejas.

Desde el punto de vista de sus aplicaciones, el procesamiento de nubes de puntos 3D es crucial en la percepción del entorno en multitud de escenarios, como por ejemplo, robótica y vehículos autónomos. *Multi-View 3D Networks (MV3D)* [13] combina vistas 2D y 3D para detectar objetos en escenas complejas. Asimismo, *PIXOR* [14] adopta un enfoque que proyecta nubes de puntos en vista de pájaro ("bird's-eye view") y utiliza CNNs para la detección de objetos, mostrando gran potencial en aplicaciones de conducción autónoma.

1.3 Casos de uso

Para acotar el alcance del trabajo, se seleccionará una escena del conjunto de datos DALES sobre la cual se tratará de reconocer los edificios que en ella se encuentran. Concretamente, la escena de partida es la presentada en el apartado anterior 1.1.

Como el tamaño de la escena es bastante grande, se utilizarán dos recortes de distinto tamaño que servirán para agilizar tanto la construcción como la comprobación de los resultados del algoritmo a cada paso. Estos recortes también serán referenciados en este documento, presentando mediciones y comparaciones relativas a su procesamiento para ofrecer evidencia sobre la evolución del rendimiento del programa. El área de menor tamaño se empleará durante la primera fase de desarrollo de la versión secuencial del algoritmo, y la de mayor tamaño durante la fase de paralelización. En la figura 1.2 se muestra el encuadre del área de tamaño menor.

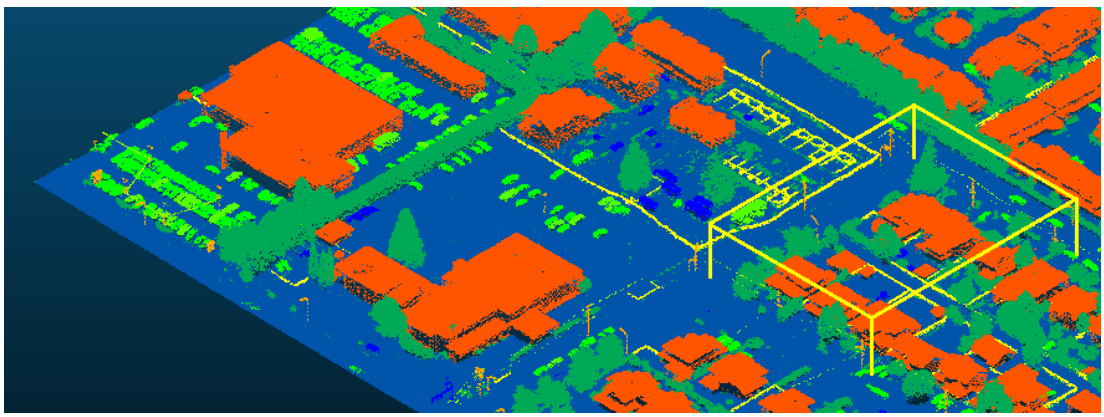


Figura 1.2: Encuadre de la escena segmentada para pruebas

Es intuitivo, pero hay que notar que en esta vista clasificada los edificios están coloreados en rojo. Es importante tener en cuenta el detalle de cada uno de ellos a la hora de contrastar esta imagen con los resultados obtenidos como salida del programa construido.

Estas escenas adaptadas se utilizarán en conjunto con otros dos recortes que servirán como objetos a identificar. Se trata de dos variaciones del edificio que se muestra en la figura 1.3 y que no está encuadrado exactamente en los dos recortes de escena, pero sí en el escenario principal. A estos dos recortes del edificio se les aplica un proceso para aislarlos y limpiarlos de puntos "anómalos". Con el objetivo de comprobar el comportamiento y la precisión de los resultados del programa, uno de los recortes contendrá el edificio por completo, y otro, únicamente el tejado.

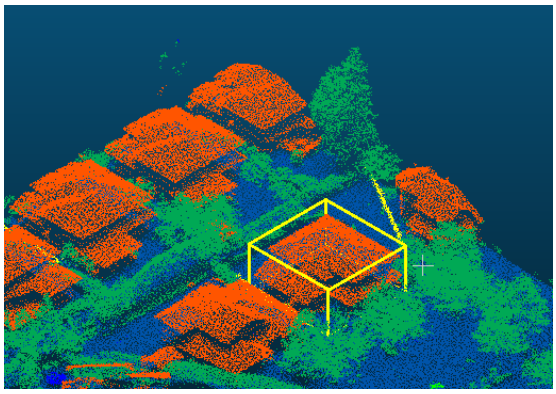


Figura 1.3: Vista del objeto a identificar

En la tabla 1.1, se indican las características de las escenas y objetos utilizados.

Rol	**Nombre**	**Área**	**Número de puntos**
Escena (Principal)	5080_54400	$500m. \times 500m.$	12,219,779
Escena (Grande)	5080_54400_big_segment	$153m. \times 184m.$	969,695
Escena (Pequeña)	5080_54400_small_segment	$92m. \times 73m.$	257,548
Objeto (Edificio)	5080_54400_house	$15m. \times 15m.$	9,227
Objeto (Tejado)	5080_54400_house_roof	$15m. \times 15m.$	8,558

Cuadro 1.1: Características de las escenas y objetos

Para buena parte de las explicaciones ofrecidas en este documento se tomará como referencia la escena (*5080_54400_small_segment*), el objeto (*5080_54400_house*), y una configuración modesta con tamaño de bloque *1.0* y *8* rotaciones, estableciendo de este modo una **línea base** que permita comparar fácilmente la evolución de rendimiento. En los experimentos de medición finales, presentados en los apartados 2.5 para las versiones secuenciales y 3.1.1 y 3.1.3 para las versiones paralelizadas, se enfrentarán respectivamente las escenas pequeña y grande con los dos objetos de búsqueda listados en la tabla 1.1, y utilizando tamaños de bloque y número de rotaciones adecuados para las capacidades de cada versión. Los valores ofrecidos para consumos de tiempo y memoria serán la media observada en 5 ejecuciones con cada una de las configuraciones estudiadas.

1.4 Métricas

Una vez construido el algoritmo y sus versiones optimizadas, se utilizarán las siguientes métricas para compararlas [15]:

- Tiempo de ejecución (T), en milisegundos
- Aceleración (S), que compara el tiempo de ejecución de la versión sin optimizar (T_s) con el tiempo de ejecución de la versión optimizada (T_p)

$$S = \frac{T_s}{T_p} \tag{1.1}$$

- Eficiencia (E), que compara la aceleración y el número de hilos de ejecución (n) al emplear paralelización con OpenMP

$$E = \frac{T_s}{T_p \cdot n} = \frac{S}{n} \tag{1.2}$$

- Porcentaje de mejora (P), que determina la reducción porcentual de tiempo de ejecución entre una versión sin optimizar (T_s) y una versión optimizada (T_p)

$$P = \frac{T_s - T_p}{T_s} \cdot 100 \tag{1.3}$$

1.5 Infraestructura computacional

El equipo actualmente en servicio en el Centro de Supercomputación de Galicia [16], el FinisTerrae III [17], con una capacidad de cómputo de 4,36 PetaFLOPS, está compuesto por 357 nodos interconectados mediante una red de baja latencia Mellanox Infiniband HDR. Cuenta con 714 procesadores Intel Xeon Ice Lake 8352Y con 32 cores a 2.2Ghz (22.848 cores) y 157 GPUs (141 Nvidia A100 and 16 Nvidia T4), 118 TB de memoria y 5.000 TB de almacenamiento de altas prestaciones Lustre.

Los nodos ILK utilizados para el presente trabajo tienen las siguientes características:

- 2x Intel Xeon Ice Lake 8352Y con 32 cores cada uno (64 cores por nodo)
- 256GB de memoria RAM (247GB para uso real)
- 960GB SSD NVMe para almacenamiento local
- 1 conexión Infiniband HDR 100

Capítulo 2

Construcción del algoritmo secuencial

Una parte crucial en el desarrollo de este trabajo ha sido comprender los conceptos matemáticos involucrados en el mecanismo de convolución, trasladar esos principios a un contexto tridimensional, y construir una primera versión de programa secuencial capaz de efectuar el cálculo de la convolución y producir resultados coherentes. Este proceso ha supuesto un buen número de iteraciones de diseño, desarrollo y pruebas, aun partiendo de un objetivo bien definido. Las diferentes posibilidades a la hora de representar la nube, tomando en consideración el espacio de almacenamiento que necesitan y la velocidad que alcanzan, y la sutileza necesaria a la hora de comparar dos nubes de puntos para determinar su similitud, también han sido desafíos importantes.

En este capítulo se dejará constancia de todas estas cuestiones, partiendo de una aproximación teórica al cálculo de convoluciones y la posterior traducción del mecanismo a pseudocódigo. Posteriormente, se realizará una breve exploración de las estructuras de datos consideradas, describiendo sus características y problemas, y ofreciendo observaciones sobre su impacto en el rendimiento. Finalmente, se presentarán los resultados y mediciones de rendimiento obtenidos para las versiones finales del algoritmo, una por cada estructura de datos considerada, después de enfrentarlas a los juegos de prueba descritos en el apartado 1.3.

2.1 Fundamentos y descripción del algoritmo

En esta sección se realizará una inmersión en el concepto de convolución, herramienta matemática fundamental para abordar el problema que se plantea, y se explorarán las variantes que resultarán de utilidad. A continuación, se generalizarán estos conceptos para definir un mecanismo de convolución para el dominio tridimensional. Finalmente, se realizará una aproximación genérica en pseudocódigo al algoritmo que se implementará.

2.1.1 Convoluciones

La **convolución** es una operación matemática, simbolizada por $*$, que combina dos funciones para describir la superposición entre ambas. Se define[18] para $t \geq 0$ como sigue:

$$(f * g)(t) = \int_0^t f(\tau)g(t-\tau)d\tau \tag{2.1}$$

En términos más simples, la convolución de dos funciones se obtiene tomando una de las funciones, digamos g, y "deslizándola" sobre la otra función, f. En cada posición, se multiplican los valores de las dos funciones en esa posición y se suman los productos. Esto se repite para todas las posiciones posibles de g sobre f, y el resultado es la función convolucionada.

Las **convoluciones discretas** son una versión de convolución que se utiliza para señales y sistemas discretos, es decir, aquellos que están definidos en puntos separados en el tiempo o el espacio. Se definen de manera similar a la convolución continua, pero con algunas diferencias importantes. En lugar de una integral, se utiliza una suma para combinar los valores de las dos funciones. Además, las funciones discretas solo están definidas en un número finito de puntos, por lo que la convolución también se define solo para un número finito de puntos. La convolución discreta de dos secuencias o vectores $f[n]$ y $g[n]$ se define[19] como:

$$(f * g)[n] = \sum_{m=0}^{N-1} f[m]g[n-m] \tag{2.2}$$

En otras palabras, para calcular el valor de $(f * g)[n]$, se multiplica cada valor de $f[m]$ por el valor correspondiente de $g[n-m]$, se suman los productos y se repite el proceso para todos los valores posibles de m.

La **convolución basada en distancias** se puede entender como una extensión de la convolución discreta en la que, para cada posición de origen n, se calcula la diferencia absoluta entre los valores correspondientes de f y g en el entorno centrado en n, y luego se promedian las diferencias. Este nuevo tipo de convolución se puede definir con las siguientes ecuaciones:

$$(f * g)'[n] = \sum_{m=0}^{N-1} |f[m] - g[n-m]| \tag{2.3}$$

$$(f * g)[n] = \frac{(f * g)'[n]}{N} \tag{2.4}$$

Este tipo de convolución puede ser más útil que las basadas en productos cuando, por ejemplo, los datos representan características de imágenes, ya que la diferencia absoluta puede ser una mejor manera de comparar las características que la multiplicación.

En la siguiente sección, se aborda la generalización del concepto de convolución basado en distancias para un espacio tridimensional representado por una nube de puntos 3D.

2.1.2 Convoluciones en tres dimensiones, definición del problema

Partiendo de una nube de puntos 3D de entrada $P_{in} \in \mathbb{R}^{m_{in} \times 3}$, se quiere obtener una nube de puntos 4D de salida $P_{out} \in \mathbb{R}^{m_{out} \times 4}$, donde $m_{out} = m_{in}$. La cuarta dimensión se utilizará para asignar un valor de disimilitud según el cual, a mayor valor, es menos plausible que en esa región del espacio se encontrase el objeto buscado.

En otras palabras, la **nube de puntos** P_{in} es una matriz en la que cada una de sus m_{in} filas tiene 3 columnas que representan las coordenadas x, y, z de un punto en un espacio tridimensional. Mientras que la matriz de salida P_{out}, de igual tamaño, tendrá una cuarta columna d que albergará la medida de disimilitud con el objeto buscado.

Para abordar el problema, se asume que se dispone de una **malla de prismas rectangulares** que representa la escena y que puede ser vista como un tensor-3 [20] tal que $G \in \mathbb{R}^{m_x \times m_y \times m_z}$. Cada elemento del tensor g_{ijk} es el conjunto de los m_{ijk} puntos de la nube de entrada P_{in} que se encuentran dentro de los límites del prisma rectangular correspondiente. El tamaño que tomarán estos prismas será uno de los parámetros de entrada del algoritmo.

Dado que cada elemento del tensor es un conjunto de puntos 3D, se puede definir el **centroide** de cada prisma en la malla $\mu_{ijk} \in \mathbb{R}^3$ de la siguiente forma

$$\mu_{ijk} = \frac{1}{m_{ijk}} \sum_{t=1}^{|m_{ijk}|} g_{ijkt} \tag{2.5}$$

Esta ecuación se utilizará para *caracterizar* a cada prisma y será el valor considerado para determinar la similitud entre dos prismas.

La **vecindad** de radio r de los elementos g_{ijk} del tensor, $N_r(G, i, j, k)$, se define como el conjunto de centroides vecinos tal y como se muestra en la siguiente ecuación

$$N_r(G, i, j, k) = \bigcup_{a=i-r}^{i+r} \bigcup_{b=j-r}^{j+r} \bigcup_{c=k-r}^{k+r} \{\mu_{abc}\} \tag{2.6}$$

Por tanto hablamos de una función que, dada una malla de prismas y los índices de uno de ellos, devuelve la vecindad de dicho prisma en la malla tomando r vecinos en cada sentido de cada eje. En general, el máximo de elementos en la vecindad es $(2r + 1)^3$.

El **kernel** Q representará al objeto buscado y una forma efectiva de obtenerlo sería, por ejemplo, haciendo un recorte de un objeto de interés de una nube de puntos, si bien, podría ser sintetizado con cualquier programa de edición de nubes de puntos. En cualquier caso, esta entrada se tratará con las mismas técnicas que se han venido describiendo, será descompuesto en una malla de prismas rectangulares -de tamaño igual a los prismas de la descomposición de la escena- y se tomará el centroide de cada prisma.

Así, el kernel se define como una vecindad de radio r, un conjunto de como máximo

$(2r+1)^3$ puntos $Q = \{q_1, \ldots, q_{(2r+1)^3}\}$. El tamaño de r vendrá determinado por el tamaño del recorte y el tamaño de prisma indicado como parámetro de entrada para la ejecución de la convolución.

Por cada kernel existen K versiones, $Q_1, \ldots, Q_k$, donde $Q_1 = Q$, que corresponden a **rotaciones** de la original. Cada rotación del kernel puede definirse según la siguiente ecuación

$$Q_t = c + (Q' - c) \cdot R\left(\frac{t-1}{K} 2\pi\right) \tag{2.7}$$

En esta ecuación Q' es la expresión matricial del conjunto de puntos de Q tal que cada fila es un punto y el número de columnas coincide con la dimensionalidad del espacio asociado. Además, c es el punto central de la malla en la que se descompone el kernel. La suma de una matriz y un vector se define como sumar el vector a cada fila de la matriz. Por último, asumiendo el eje *z* como el eje vertical, $R(\theta)$ corresponde a la siguiente ecuación

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

El resultado de la convolución de una vecindad $N_r(G, i, j, k)$ con el kernel Q se define como la distancia entre la vecindad y la rotación del kernel que la minimiza, tal y como se expresa en la siguiente ecuación

$$\delta(N_r(G, i, j, k), Q) = \min_t d(N_r(G, i, j, k), Q_t) \tag{2.9}$$

La distancia entre dos conjuntos de puntos $d(A, B)$, ya que tanto $N_r(G, i, j, k)$ como Q_t lo son, se define en este caso como el promedio de las distancias entre los puntos de igual posición en la malla que conforma cada conjunto. Se expresaría con la siguiente ecuación

$$d(A, B) = \frac{1}{|B|} \sum_{\substack{a_{lmn} \in A \\ b_{lmn} \in B}} ||a_{lmn} - b_{lmn}|| \tag{2.10}$$

Para que la convolución de una vecindad $N_r(G, i, j, k)$ y un kernel Q funcione, primero debe realizarse una alineación de sus sistemas de coordenadas. Una forma conveniente de hacerlo es, para ambos conjuntos, restar a cada punto el vértice mínimo de la malla que lo contiene.

Para obtener el resultado de la convolución entre la nube de puntos y el kernel Q, se realiza la convolución de cada vecindad de tamaño r que contiene la malla de prismas rectangulares formada a partir de la nube de puntos inicial y el kernel Q. A cada prisma quedará asociado el valor mínimo de disimilitud obtenido en cualquiera de las convoluciones en las que participa.

En la figura 2.1 se exponen de manera visual las ideas esenciales del procedimiento descrito.

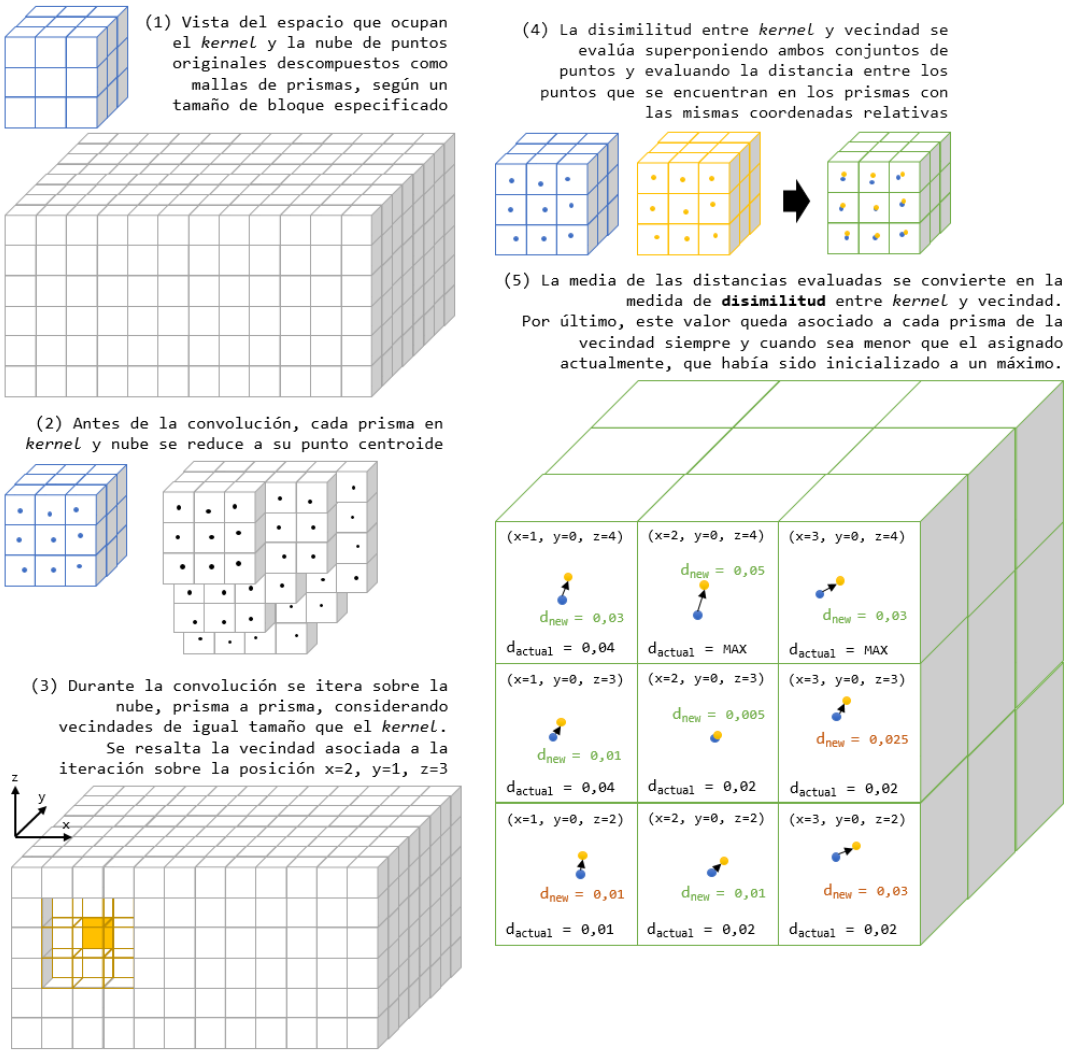


Figura 2.1: Representación de los pasos fundamentales del algoritmo de convolución

2.1.3 Pseudocódigo

En el procedimiento 1 se describe el algoritmo de carga de una nube de puntos en memoria. Hay que notar que, para poder acomodar la nube en una estructura de datos adecuada, resulta imprescindible determinar sus límites, por lo que inicialmente se realiza la carga de los puntos que conforman la nube en una estructura auxiliar tipo colección. Conociendo los límites del espacio que ocupa la nube, y el tamaño de bloque parametrizado, ya es posible

evaluar la dimensión de los prismas de la malla en la que se debe descomponer la nube. En este pseudocódigo no se predetermina una estructura de datos concreta, -en la sección 2.2 dedicada a estructuras de datos se analizarán varias posibilidades-, pero, como premisa, debe proveer un acceso a conjuntos de puntos vecinos según las coordenadas de la malla.

Algorithm 1 Carga de fichero de nube de puntos

```
procedure LOAD(cloud_file, block_size)
    points ← LOAD_FILE(cloud_file)                          ▷ Carga en una colección de puntos
    limits ← GET_LIMITS(points)                                              ▷ Evalúa límites
    dim ← EVAL_DIM(limits, block_size)                  ▷ Evalúa la dimensión de la malla de prismas
    structure ← CREATE_POINTS_STRUCTURE(dim)      ▷ Crea una estructura de datos que permita
acceso por las coordenadas de la malla, cada elemento albergará una colección de puntos
    LOAD_POINTS_INTO_STRUCTURE(points, structure, dim)  ▷ Traspasa los puntos a la estructura
final
    return {dim = dim, data = structure}                ▷ Tipo estructurado con dimensión y datos
end procedure
```

El procedimiento 2 detalla la transformación de la estructura de datos que alberga una nube de puntos de entrada en una nueva estructura de igual dimensión pero albergando únicamente el centroide de cada prisma de la malla en la que se descompuso la nube. Este paso es muy conveniente para reducir los tiempos de ejecución, ya que permite una caracterización concisa de cada vecindad evaluada.

Algorithm 2 Crea una nueva estructura que alberga los centroides de cada elemento de la malla

```
procedure REDUCE_TO_CENTROIDS(cloud)
    dim ← cloud.dim
    structure ← CREATE_POINT_STRUCTURE(dim)      ▷ Crea una estructura de datos que permita
acceso por las coordenadas de la malla, cada elemento alberga un único punto
    for x ← 0 to cloud.dim.x do
        for y ← 0 to cloud.dim.y do
            for z ← 0 to cloud.dim.z do
                structure[x][y][z] ← CENTROID(cloud.data[x][y][z])      ▷ Evalúa el centroide de la
colección de puntos
            end for
        end for
    end for
    return {dim = cloud.dim, data = structure}      ▷ Tipo estructurado con dimensión y datos
end procedure
```

El procedimiento 3 detalla la evaluación de la disimilitud entre una sección concreta de la escena, la *vecindad*, y un objeto de interés, el *segmento* que representa el *kernel*. Tanto la vecindad como cada una de las rotaciones del segmento que se envían a este procedimiento son estructuras que representan las mallas de prismas ya reducidas a sus centroides. Por simplicidad, se considera la iteración sobre las coordenadas locales de la vecindad y del kernel,

que son de igual dimensión, pero se podría considerar la vecindad como una mera ventana con coordenadas relativas dentro de la escena general. Como medida de disimilitud, en este procedimiento se evalúa la media de las distancias entre los centroides de cada prisma de la vecindad y del segmento en coordenadas análogas.

Algorithm 3 Evalúa la disimilitud entre una vecindad y un segmento

```
procedure NEIGHBORHOOD_CONVOLUTION(neighborhood, segment)
    a ← 0                                                    ▷ Acumulador
    c ← 0                                                    ▷ Contador
    for i ← 0 to segment.dim.x do
        for j ← 0 to segment.dim.y do
            for k ← 0 to segment.dim.z do
                p_n ← neighborhood.data[i][j][k]             ▷ Punto centroide
                p_s ← segment.data[i][j][k]                  ▷ Punto centroide
                a ← a+DISTANCE(p_n, p_s)                     ▷ Evaluar distancia entre los puntos
            end for
        end for
    end for
    return a ÷ c
end procedure
```

Por último, en el procedimiento 4 se describe el funcionamiento del programa en general y de los bucles de convolución en particular.

En la parte inicial se realizan las cargas en memoria de las nubes de puntos que representan la escena y el objeto buscado (segmento), produciendo una estructura de tipo malla de prismas que tienen asociados todos los puntos que se encuentran dentro de los límites particulares de cada prisma. Estas estructuras originales se transforman en otras de igual dimensión de modo que cada prisma tenga ya únicamente asociado su centroide. Para la estructura correspondiente al segmento se crea una serie de copias rotadas sobre el eje z en los ángulos apropiados según el número de rotaciones demandadas. Con esta preparación previa de las estructuras de datos ya se procede a efectuar la convolución.

Durante la convolución se itera prisma a prisma la malla en la que se descompuso la escena original. Para cada uno de los prismas, se consideran a su vez todas las rotaciones del segmento original, teniendo en cuenta que cada una de las rotaciones obtenidas tendrá su propia dimensión. Con la dimensión específica de cada segmento rotado, y las coordenadas del prisma central, se puede evaluar la vecindad de la escena original que se debe contemplar en cada iteración. A continuación, se evalúa la disimilitud entre esta sección y el segmento rotado. La estructura que representa cada punto en el espacio tridimensional ofrece un espacio para almacenar este valor de salida, que, durante su creación, se inicializa al máximo valor almacenable según el tipo de dato escogido para su representación, normalmente de punto flotante. Por simplicidad, y ya que los centroides pueden considerarse representantes de todos los puntos vecinos en sus correspondientes prismas, este dato queda asociado temporalmente

a ellos. Siempre y cuando, eso sí, el nuevo valor obtenido sea menor que el que contiene actualmente. Este paso constituirá una sección crítica en el algoritmo cuando se introduzca paralelismo, ya que las vecindades de los prismas tienen solapes, por tanto, son susceptibles de sufrir accesos concurrentes en zonas de trabajo contiguas asignadas a distintos hilos.

Una vez completada la convolución, en los pasos finales del algoritmo, se reasignan los valores de disimilitud que almacena cada centroide a sus puntos vecinos que quedaron asociados en la estructura de datos original que representa la escena. Por último, ya sólo queda escribir la nube de puntos en el archivo de salida, incluyendo ahora para cada punto su correspondiente valor de disimilitud, tal y como se estipulaba en las especificaciones.

Algorithm 4 Algoritmo de cálculo de convolución

procedure CONVOLUTION($cloud\_file$, $segment\_file$, $block\_size$, $num\_rotations$, $cloud\_out\_file$)
    ▷ Cargar y preprocesar las estructuras de datos
    $cloud\_original \leftarrow$ LOAD($cloud\_file$, $block\_size$) ▷ Estructura de colecciones de puntos
    $segment\_original \leftarrow$ LOAD($segment\_file$, $block\_size$) ▷ Estructura de colecciones de puntos
    $cloud \leftarrow$ REDUCE_TO_CENTROIDS($cloud\_original$) ▷ Estructura de centroides
    $segment \leftarrow$ REDUCE_TO_CENTROIDS($segment\_original$) ▷ Estructura de centroides
    $segment\_rotations \leftarrow$ ROTATE($segment$, $num\_rotations$) ▷ Lista de estructuras de centroides

    ▷ Realizar la convolución
    for $x \leftarrow 0$ to $cloud.dim.x$ **do**
        for $y \leftarrow 0$ to $cloud.dim.y$ **do**
            for $z \leftarrow 0$ to $cloud.dim.z$ **do**
                for $r \leftarrow 0$ to $num\_rotations$ **do**
                    $s \leftarrow segment\_rotations[r]$ ▷ Estructura de centroides
                    $n \leftarrow$ NEIGHBORHOOD($cloud$, x, y, z, $s.dim$) ▷ Obtener la vecindad como estructura de centroides
                    $d \leftarrow$ NEIGHBORHOOD_CONVOLUTION(n, s) ▷ Evaluar la disimilitud

                    ▷ Para cada elemento en la vecindad, se asigna la disimilitud calculada siempre que sea menor que el valor actual
                    for $i \leftarrow 0$ to $s.dim.x$ **do**
                        for $j \leftarrow 0$ to $s.dim.y$ **do**
                            for $k \leftarrow 0$ to $s.dim.z$ **do**
                                $n.data[i][j][k].d \leftarrow$ MIN($n.data[i][j][k].d$, d) ▷ ¡Sección crítica!
                            end for
                        end for
                    end for
                end for
            end for
        end for
    end for

    ▷ Se vuelcan los datos de disimilitud sobre la estructura original
    for $x \leftarrow 0$ to $cloud.dim.x$ **do**
        for $y \leftarrow 0$ to $cloud.dim.y$ **do**
            for $z \leftarrow 0$ to $cloud.dim.z$ **do**
                for $p \in cloud\_original.data[x][y][z]s$ **do**
                    $p.d \leftarrow cloud.data[x][y][z].d$
                end for
            end for
        end for
    end for

    WRITE($cloud\_original$, $cloud\_out\_file$) ▷ Se escribe la nube de salida
end procedure

2.2 Estructuras de datos

El primer reto que se afronta en el desarrollo del programa es la selección de una estructura de datos apropiada para representar en memoria la gran cantidad de datos que conforma una nube de puntos. La primera intuición, al tratarse de información organizada de forma tridimensional, es la de utilizar una estructura de array tridimensional. Sin embargo, al observar una nube de puntos, es fácil comprobar la cantidad de espacio que se desperdiciaría debido a la existencia de numerosas zonas sin información. Esto provoca que, por respeto al tamaño del problema, se consideren de inicio otro tipo de representaciones que hagan un uso de memoria menos extensivo. Sin embargo, ya en este momento se puede desvelar que la elección de una solución adecuada para este problema ha sido un viaje de ida y vuelta. En los siguientes apartados se describirán brevemente las alternativas valoradas y los problemas encontrados.

Tal y como se propone en el pseudocódigo del apartado anterior, lo que se busca es una estructura que permita no sólo representar los puntos 3D que conforman la nube, sino también acceder de manera eficiente a los puntos dentro de una región específica -recordemos que es necesario descomponer la nube en una malla de prismas o «bloques»-, así como a la vecindad de cada una de estas regiones, es decir, a los bloques adyacentes dentro de un radio determinado. Además, la estructura deberá ser flexible para contener distintos tipos de datos y facilitar las transformaciones propuestas en el pseudocódigo.

2.2.1 Árboles octales

Una estructura que se adapta de forma muy natural y eficiente a la representación de información en tres dimensiones son los árboles octales (octree). El árbol octal organiza los datos en una estructura jerárquica en la que cada nodo representa un cubo tridimensional (octante) que puede dividirse en ocho subcubos de igual tamaño.

Esto es bastante eficiente en términos de espacio ya que sólo se representan los octantes que contienen información, mientras que los octantes vacíos no ocupan espacio. Y también es bastante eficiente en términos de búsqueda a la hora de determinar en qué octantes se encuentran los objetos o puntos de interés.

En general, la complejidad temporal de buscar en un árbol octante se estima como $O(\log n)$, donde n es el número de nodos. Esto significa que el tiempo de búsqueda crece de manera muy lenta a medida que aumenta el número de nodos.

A la hora de llevarlo a la práctica para el cálculo de convoluciones, el árbol octal plantea un desafío notable para conseguir un modelo compatible con las necesidades estipuladas en el pseudocódigo propuesto en el apartado 2.1.3, en el que se propone una interfaz que facilite el acceso eficiente a regiones delimitadas de la nube de puntos. Contrariamente a lo que se podría intuir inicialmente, la estrategia de uso no ha sido la de construir un árbol octal debidamente

balanceado en el que se organicen directamente los puntos pertenecientes a una nube, sino en emplear el árbol octal para representar específicamente las regiones del espacio en las que se necesitaba dividir la nube, según el tamaño de bloque estipulado.

La construcción de un árbol octal se realiza dividiendo de forma recursiva el espacio tridimensional en octantes cada vez más pequeños hasta alcanzar la granularidad deseada. En este caso, se pretende dividir el espacio hasta obtener cubos de lado igual al tamaño de bloque indicado como parámetro del programa. Para conseguir compatibilizar la nube con este requisito, lo que se hace es contemplar un área de tamaño suficiente para contener la nube original y que la subdivisión recursiva finalice en bloques del tamaño exacto. Aquí se ha optado por considerar una malla cúbica con lado $l = 2^{\lceil \log_2 max(dim.x, dim.y, dim.z) \rceil}$. De este modo, con la raíz del árbol asociada al bloque central de la malla tridimensional, será posible direccionar toda la malla y como máximo será necesario alcanzar la profundidad $\lceil \log_2 max(dim.x, dim.y, dim.z) \rceil$ para encontrar un determinado bloque. Debido a que el espacio que representa el árbol octal es más grande que el de la malla en la que se descompone la nube, y da lugar a un espacio de direcciones más amplio, es posible realizar un traslado de coordenadas de modo que la malla quede centrada en la raíz del árbol. Esto permitirá que las búsquedas se comporten como si el árbol estuviese perfectamente balanceado. Sin embargo, también es factible utilizar las coordenadas originales de la malla, aunque implicará iteraciones extra durante las búsquedas.

Por otro lado, la naturaleza recursiva inherente de los árboles octales, también plantea inicialmente dudas sobre cómo se podría abordar posteriormente su paralelización. Una operación recursiva no es buena candidata para paralelizar con OpenMP ya que éste está potenciado para la paralelización de bucles y operaciones iterativas para las que es necesario conocer el número de iteraciones de forma previa a su ejecución. Debido a ello, resulta más sencillo pensar en estrategias de división del problema inicial para repartir el cómputo entre varios nodos, que no en acelerar las operaciones en las que se divide la solución. Esta idea resultará ser una buena estrategia de optimización independientemente de la estructura de datos que se evalúe.

La implementación de convolución realizada con árboles octales con las características aquí descritas resultó tener un desempeño mediocre, necesitando un tiempo medio de 3 minutos y 30 segundos para procesar el caso descrito como línea base en 1.3. Estando perfectamente balanceado, el árbol octal requiere $\log_8 n$ iteraciones de media para acceder a sus elementos, y esto empuja a pensar en alternativas que ofrezcan un acceso más directo. De aquí, surge la idea de utilizar una nueva estructura: el mapa.

2.2.2 Mapas

Un mapa, matriz asociativa o tabla de dispersión, es una estructura de datos que almacena pares de clave y valor, donde cada clave es única y está asociada a un único valor. Es una

estructura específicamente diseñada para permitir un acceso eficiente a los valores utilizando sus claves correspondientes. En este tipo de estructuras la búsqueda se realiza en un tiempo constante $O(1)$, lo que significa que no dependería del tamaño del mapa.

Se elabora una implementación de mapa con una función de dispersión basada en las coordenadas de cada bloque. Para gestionar posibles colisiones, cuando la función de dispersión produzca la misma posición de almacenamiento para dos o más coordenadas, se utiliza una estrategia de encadenamiento, es decir, se mantiene una lista con todos los elementos con el mismo índice. Esto conlleva una ligera pérdida de rendimiento en tales ocasiones. Para tratar de mitigar estas situaciones de degradación, también se introduce una estrategia de rehashing por la cual, superado un determinado factor de carga (75%) [1], se dispara un subproceso que amplía (duplica) la capacidad del mapa y vuelve a evaluar las posiciones de almacenamiento de cada bloque. Todo esto sucede durante la carga de nubes de puntos en memoria, por lo que no tiene influencia a la hora de calcular las convoluciones.

Tal y como se propone en el pseudocódigo en el apartado 2.1.3, y de forma análoga a la implementación con árboles octales, la implementación de mapa realizada permite flexibilidad en el tipo de datos de los valores almacenados. Durante la carga de la nube en cada nodo valor del mapa se almacena una lista con los puntos cuyas coordenadas se encuentran en el área delimitada para cada bloque de la malla que se está representando. Posteriormente, a partir de estos mapas de listas de puntos, se construyen mapas de puntos con los centroides de cada bloque, necesarios para evaluar la convolución del modo propuesto.

El diseño de la función de dispersión resulta crucial para que las búsquedas sobre el mapa ofrezcan un rendimiento adecuado. En los experimentos iniciales, una sencilla función $(x + y + z)$ mod $capacity$ conlleva a un alto índice de colisión de las claves obtenidas y un pobre rendimiento, por lo que resulta necesario efectuar algunas iteraciones de desarrollo buscando una función con un mejor comportamiento. Dadas las limitaciones de tiempo y lo extenso de la problemática, durante la realización de este trabajo no se aborda un análisis exhaustivo de las funciones de dispersión óptimas para este caso. En su lugar, tras considerar algunas variantes, se selecciona una estrategia simplificada pero razonablemente efectiva, que consiste en agregar un factor de sesgo que ayude a reducir la agrupación de claves. En la nueva función de dispersión construida, cada coordenada es multiplicada por un número primo de 5 dígitos $(x \cdot prime1 + y \cdot prime2 + z \cdot prime3)$ mod $capacity$.

Así, los resultados obtenidos para la ejecución de esta nueva versión del algoritmo mejoran significativamente los del algoritmo basado en árboles octales, siendo ahora necesarios de media 2 minutos y 5 segundos para procesar el mismo caso base, lo que supone una mejora

[1] La elección de un factor de carga por defecto del 75% se fundamenta en las observaciones de otras implementaciones, como la de HashMap de Java. Como regla general, el factor de carga del 75% ofrece un buen equilibrio entre los costes de tiempo y espacio. Valores más altos disminuyen la sobrecarga de espacio pero aumentan el coste de búsqueda. [21]

del 40% en el tiempo de ejecución. Aun teniendo una función de dispersión imperfecta, con un índice de colisiones bajo, pero notable, para el mismo tamaño de problema, el número de iteraciones necesarias para encontrar los elementos con este algoritmo es menor en comparación con el basado en árboles octales, y esto es lo que se refleja en los tiempos obtenidos.

Un perfilado de rendimiento del código clarifica que el 99% del tiempo consumido se emplea en la función de búsqueda de elementos en el mapa. Tal circunstancia sugiere la necesidad de explorar estrategias que permitan reducir el número de búsquedas. Estas primeras versiones ingenuas basadas en árboles octales o mapas no introducen apenas mejoras algorítmicas, y ciertos aspectos se hacen ahora evidentes, como por ejemplo, la repetición de búsquedas de bloques de las vecindades de la escena durante el procesamiento de las rotaciones del objeto buscado. Para abordar esta problemática, sería beneficioso contar con una estructura caché que sea capaz de proveer un acceso más directo sin tener que repetir las operaciones de búsqueda.

Esta necesidad invita a pensar, de nuevo, en arrays tridimensionales. Se ensaya todavía sobre estos algoritmos la introducción de un array caché para el trabajo con las vecindades de la nube original, y sus resultados serán expuestos en el apartado 2.5. El siguiente paso natural sería transformar las estructuras de almacenamiento de las rotaciones del objeto de búsqueda en arrays tridimensionales, ya que también sufren búsquedas repetitivas en cada iteración, teniendo de este modo una solución híbrida, con árboles octales y mapas para la representación de la escena, y arrays tridimensionales para la representación del objeto de búsqueda y sus rotaciones. Resultaría interesante efectuar este paso intermedio, sin embargo, se ha optado por abordar directamente el uso de arrays tridimensionales para todos los elementos, como se detallará en el siguiente apartado.

2.2.3 Arrays

Los elementos de un array se almacenan de manera contigua en la memoria. Esto significa que cada elemento del array se encuentra adyacente al siguiente en la memoria, lo que facilita el acceso secuencial y aleatorio a los elementos. Acceder a un elemento específico de un array es una operación muy eficiente, ya que el acceso se realiza en tiempo constante $O(1)$, es decir, independientemente del tamaño del array, el tiempo necesario para acceder a un elemento específico es el mismo.

Las nubes de puntos obtenidas con LiDAR aéreo tratadas en este trabajo, se caracterizan por contener más información en las dos primeras dimensiones que en la tercera. Desde el punto de vista del rendimiento, y a pesar del desperdicio de memoria que supondría la representación de este tipo de nubes en estructuras de tamaño fijo, el array tridimensional resulta ser la mejor estructura para albergar las nubes y disponerlas para el cálculo de las convoluciones. El enfoque del algoritmo propuesto en el pseudocódigo del apartado 2.1.3 no cambia,

ya que la estructuración de la nube en bloques y su acceso por coordenadas de bloque se mantiene, por lo que la refactorización llevada a cabo es relativamente sencilla.

Para el problema establecido como caso base se obtiene una reducción de tiempo cercana al 87% respecto a la versión con mapas sin caché, empleando ahora algo menos de 17 segundos.

En cuanto al consumo de memoria, se ha observado que las diferencias son menores de lo que se esperaba inicialmente. Este fenómeno se debe a que las representaciones más artificiales, como los árboles octales y los mapas, requieren almacenar una cantidad adicional de información en sus nodos para el correcto funcionamiento de los algoritmos. Esto se evidencia por ejemplo en las coordenadas de los bloques de la malla representada, las cuales, a diferencia de los arrays, no son inherentes a la propia estructura y, por lo tanto, necesitan ser almacenadas específicamente en los nodos de árboles y mapas. Aún así, en los datos mostrados en la tabla 2.1, se puede apreciar el mayor consumo del algoritmo basado en arrays, -aunque es totalmente asumible en comparación con la mejora de rendimiento que ofrece-, y el buen comportamiento del algoritmo basado en mapas, a pesar de ser también una estructura con tendencia a desperdiciar memoria.

En la tabla 2.1 se muestran medidas de consumo de memoria tomadas para la ejecución secuencial de los tres algoritmos descritos en este capítulo con distintos parámetros de entrada. Como parámetros invariantes, y poco relevantes para esta medida, tenemos el número de rotaciones, que es 8, y el objeto buscado *5080_54400_house*. Los datos en esta tabla revelarán además que el consumo de memoria está directamente relacionado con el tamaño de la escena y la precisión deseada (el tamaño del bloque), por lo que es importante tener en cuenta que, a medida que la escena y, especialmente, la precisión aumentan, no se presenta únicamente un problema de cómputo, sino también uno de capacidad de memoria.

Escena	Tam. de bloque	Octree	ListMap	Array
5080_54400_small_segment	1,0	52	51	51
5080_54400_small_segment	0,5	58	55	61
5080_54400_big_segment	1,0	189	186	188
5080_54400_big_segment	0,5	213	202	236
5080_54400	1,0	2355	2355	2355
5080_54400	0,5	2662	2560	3072

Cuadro 2.1: Consumo de memoria en MiB de los algoritmos secuenciales para distintos casos

2.3 Mejoras algorítmicas

Todas las versiones desarrolladas, tanto las secuenciales como las paralelizadas que se presentarán en el siguiente capítulo, sufren un proceso evolutivo análogo durante el cual se ven paulatinamente beneficiadas con todas las mejoras técnicas y algorítmicas detectadas que les son de aplicación. Se pretende con esta política encontrar cierta justicia en la comparación entre las distintas versiones y delimitar correctamente las razones detrás de cada mejora de rendimiento conseguida. Lo que esto significa es que las versiones secuenciales que se han desarrollado se han intentado optimizar al máximo con las técnicas disponibles, y que las mejoras de rendimiento que se presentarán en el siguiente capítulo serán puramente basadas en la paralelización del algoritmo.

En la sección 2.2.2 dedicada a mapas se adelantaba una de estas mejoras algorítmicas, que consiste en la introducción de una estructura de datos auxiliar basada en arrays tridimensionales para ser utilizada a modo de caché en las iteraciones de la convolución. Mediante su uso, es posible evitar la repetición de búsquedas de elementos a la hora de almacenar la disimilitud evaluada y a la hora de procesar las rotaciones del objeto buscado. Sólo sería aplicable a los algoritmos basados en árboles octales y mapas, ya que la versión basada en arrays tridimensionales ya se beneficia del uso de este tipo de estructuras. Esta mejora da lugar a versiones híbridas que podrían beneficiarse de lo mejor de ambos mundos en cuanto a consumo de memoria y rendimiento, si bien, sin perder de vista el consumo de memoria, en este trabajo se ha antepuesto el rendimiento superior de la versión basada íntegramente en arrays.

Otra mejora algorítmica introducida sobre las versiones primigenias se centra en cómo evaluar la convolución en los límites del espacio demarcado por la nube. Las versiones iniciales manejaban un concepto de bloque «fuera de límites» y ejecutaban comprobaciones y lógica específica en cada iteración para manejar la situación, introduciendo además algunas condiciones de salida de las iteraciones (*break*). La mejora algorítmica consistió en eliminar esta gestión específica de las direcciones fuera de límites simplemente aplicando una estrategia *clamp*, de forma que las coordenadas de acceso a los bloques se corrigen a los rangos válidos, obteniendo siempre el bloque más cercano dentro de los límites para cualesquiera coordenadas fuera de límites. Con esto se eliminan comprobaciones y saltos en las iteraciones dando lugar a un código más claro y eficiente.

El resto de las optimizaciones reseñables se centran en la utilización de técnicas básicas de programación eficiente, prestando atención a los mecanismos de reserva de memoria para maximizar la contigüidad de las zonas mejorando la localidad de datos, realizando la declaración de variables internas de forma externa a los bucles evitando sobrecargas innecesarias, reutilizando cálculos de direccionamiento de elementos en memoria y reduciendo al máximo accesos a memoria utilizando variables auxiliares a modo de caché.

2.4 Presentación de resultados

En esta sección se presentan las salidas producidas por las distintas versiones secuenciales del algoritmo de convolución tratadas en este capítulo y con diversas parametrizaciones, no únicamente las estipuladas como línea base. En realidad, se ha hecho el esfuerzo de mantener la consistencia en todas las versiones construidas, de modo que todas producen los mismos resultados para los mismos parámetros de entrada, así que, se mostrarán únicamente los resultados de una de ellas, concretamente los producidos por la versión basada en arrays.

Se comienza por mostrar en la figura 2.2 la salida correspondiente a la escena pequeña *5080_54400_small_segment* siendo el objeto buscado *5080_54400_house*. El tamaño de bloque estipulado en este caso es de 2,00 unidades (metros), lo que significa que durante la convolución las nubes se disgregan en mallas cúbicas de 2,00 $\times$ 2,00 $\times$ 2,00 metros. Se incluyen además los resultados para 1 y 8 rotaciones del objeto buscado.

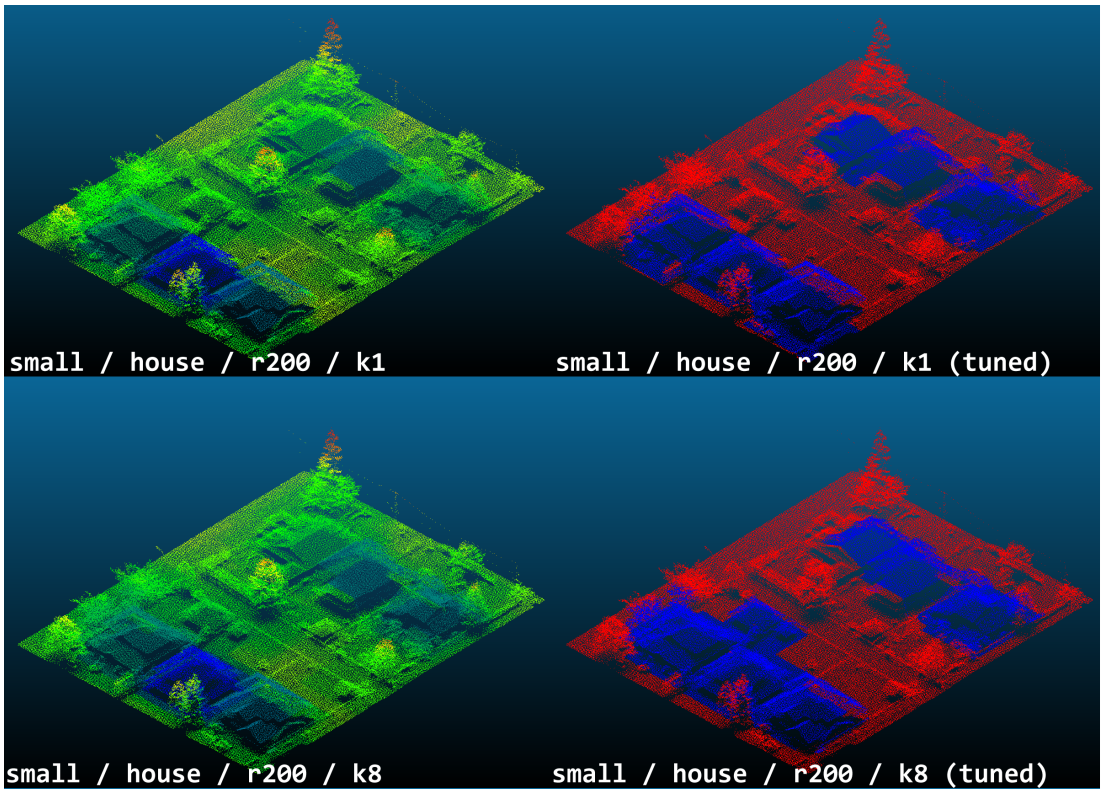


Figura 2.2: Salida para escena pequeña y edificio con tamaño de bloque 2,00

Las imágenes a la izquierda corresponden a la salida original que se colorea con una escala azul > verde > amarillo > rojo para los valores de disimilitud que van de menor a mayor. Es decir, las zonas azules corresponderían con mayor probabilidad a edificios mientras que las

zonas rojas con mayor probabilidad no corresponderían a edificios. A la derecha se muestran las salidas reajustadas tras haber reducido la escala de colores a solamente dos intervalos, situando su frontera en un valor elegido de modo que se produzca el resalte del mayor número de edificios posible, y facilitando así la apreciación de la precisión de la salida.

En cuanto a la variación en el resultado producida por el número de rotaciones del objeto buscado, aunque puede ser difícil verificarlo en las imágenes debido a la falta resolución de este medio, podría parecer que el resultado con una rotación produce resultados más precisos, ya que los edificios se muestran ligeramente mejor delimitados. Este fenómeno observado deriva del hecho de que el objeto buscado está totalmente alineado con los edificios en esta escena, y la introducción de rotaciones del objeto buscado provoca que, en cierta medida, se difuminen los bordes de los edificios. El efecto positivo de la introducción de rotaciones debería observarse en áreas con calles y edificios dispuestos con menor ortogonalidad.

En la figura 2.3 se muestra el resultado para el mismo escenario pero habiendo reducido el tamaño de bloque a 1,00 metro, pudiendo apreciarse una ligera mejora en la precisión.

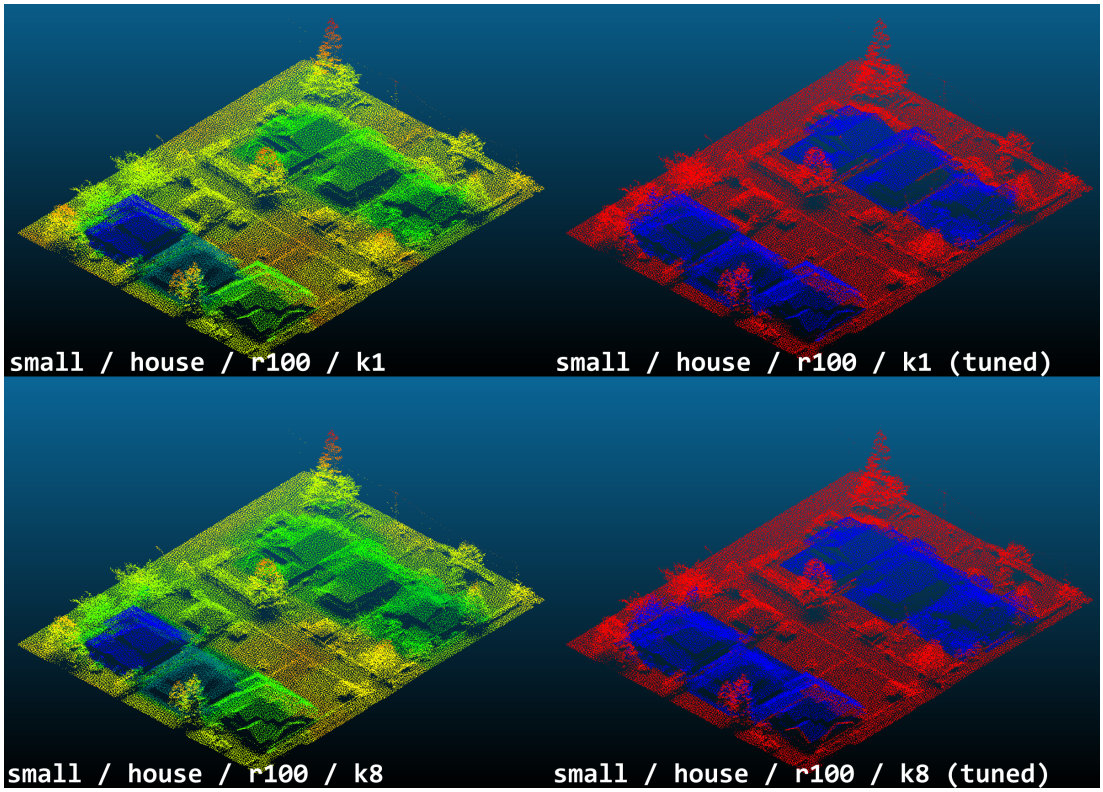


Figura 2.3: Salida para escena pequeña y edificio con tamaño de bloque 1,00

A continuación, se presentan los resultados obtenidos utilizando el objeto de búsqueda *5080_54400_house_roof*. Como se indicaba en la sección 1.3, se trata del tejado del mismo edi-

ficio que se ha utilizado en el caso anterior. En esta ocasión, en la figura 2.4 se muestra conjuntamente el resultado para las convoluciones con tamaño de bloque de 2,00 y 1,00 metros.

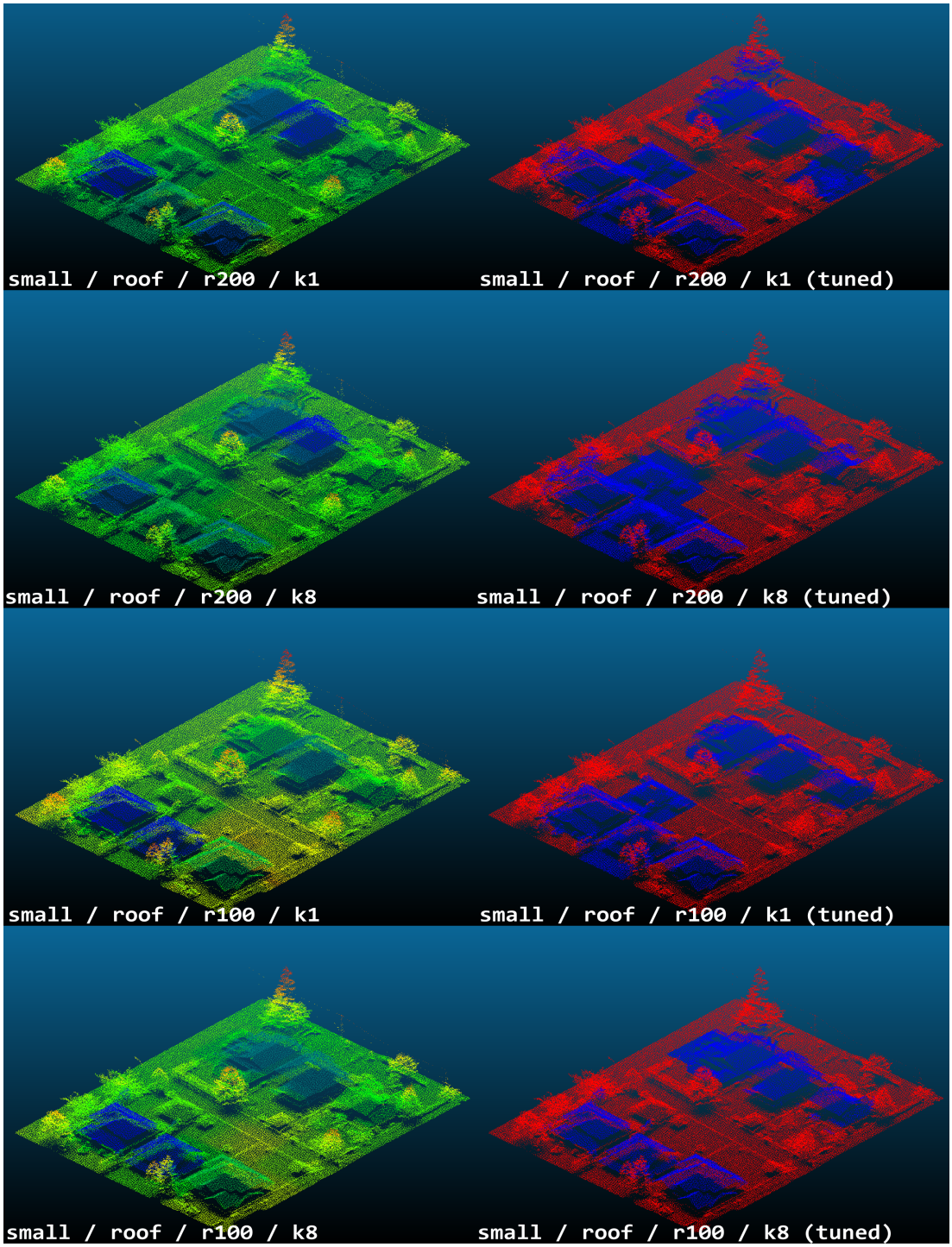


Figura 2.4: Salida para escena pequeña y tejado con tamaños de bloque 2,00 y 1,00

Como se puede observar, los resultados obtenidos al utilizar *5080_54400_house_roof* como objeto de búsqueda muestran variaciones significativas en comparación con los producidos por el objeto *5080_54400_house*. Visualmente, resulta complicado determinar si estos resultados son mejores o peores. Por un lado, en los resultados presentados en la escala de colores original, se aprecia un área mayor en tonos azul correspondiendo de forma correcta a edificios, lo que podría indicar una mejor precisión en la detección. Sin embargo, al ajustar los umbrales de la escala de colores buscando su convergencia, se observa que se pierden algunos edificios o, por el contrario, se incluyen zonas circundantes en exceso, lo que sugiere una mayor imprecisión en la detección. No obstante, esta escena es demasiado pequeña como para extraer conclusiones definitivas acerca de la idoneidad de los objetos de búsqueda utilizados.

Como se verá en la siguiente sección, una ventaja que sí presenta el uso del objeto de búsqueda *5080_54400_house_roof* frente al objeto *5080_54400_house* es que el tiempo de ejecución de la convolución es sensiblemente menor. Esto se debe a que este recorte contiene un número de puntos inferior y la convolución se realiza evaluando vecindades de menor tamaño. En general, dependiendo de la versión, tanto el tiempo del programa como el tiempo de convolución mejoran entre un 36% y un 39% utilizando este objeto de búsqueda más ajustado.

A la vista de los resultados presentados en esta sección, se puede concluir que el algoritmo produce unos resultados razonablemente correctos aun con tamaños de bloque grandes como los utilizados. Otra cuestión, que se confirmará con los datos presentados en la siguiente sección, es que el comportamiento del algoritmo, tanto en términos de precisión como de rendimiento, está estrechamente relacionado con la elección del objeto de búsqueda, es decir, depende del *kernel* utilizado.

2.5 Medidas de rendimiento

En la tabla 2.2 se comparan los tiempos de ejecución **secuencial** de las tres versiones iniciales del algoritmo de convolución. Los parámetros de ejecución invariantes para este experimento son: escena *5080_54400_small_segment* y tamaño de bloque 1,0. La tabla incluye dos parámetros variables, el objeto utilizado como *kernel* y el número de rotaciones.

		Octree		ListMap		Array	
Objeto	Rot.	Total	Convol.	Total	Convol.	Total	Convol.
house	1	19,478	18,873	13,121	12,487	2,101	1,512
house_roof	1	12,045	11,444	8,250	7,618	1,498	0,918
house	8	210,396	209,774	124,439	123,791	16,472	15,870
house_roof	8	130,950	130,206	75,755	75,111	10,396	9,802

Cuadro 2.2: Tiempos de ejecución en segundos de los algoritmos secuenciales iniciales

Tomando como referencia la versión con árboles octales, se calcula la siguiente tabla en la que se muestra la **aceleración** para los distintos algoritmos y casos de prueba.

		Octree		ListMap		Array	
Objeto	Rot.	Total	Convol.	Total	Convol.	Total	Convol.
house	1	1	1	1,48	1,51	9,27	12,48
house_roof	1	1	1	1,46	1,50	8,04	12,46
house	8	1	1	1,69	1,69	12,77	13,22
house_roof	8	1	1	1,73	1,73	12,60	13,28

Cuadro 2.3: Aceleración de los algoritmos secuenciales iniciales

En la figura 2.5 se presentan de forma gráfica los datos de la **aceleración** tabulada en 2.3.

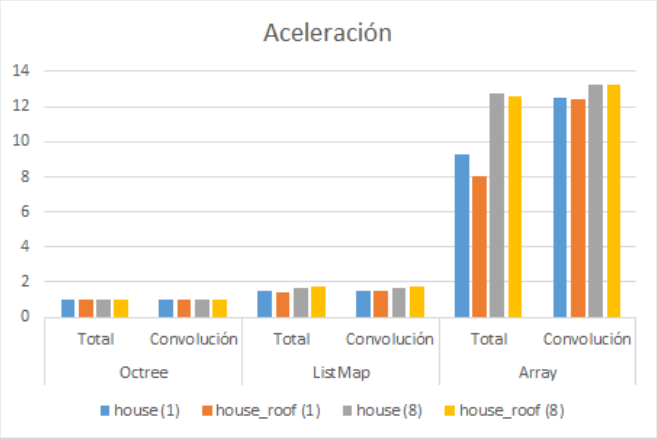


Figura 2.5: Aceleración de los algoritmos secuenciales iniciales

Los resultados presentados confirman la idoneidad de la representación basada en arrays para la ejecución del algoritmo de convolución propuesto. Esta versión simplifica enormemente el acceso a los elementos de la malla en la que se descompone la nube, ya que se reduce a calcular un direccionamiento a memoria, en contraposición a la navegación por octantes y al cálculo de una clave de dispersión. Además, se beneficia de una mejor localidad de datos debido a la disposición contigua de los mismos, en contraposición a la impredictibilidad de las otras estructuras.

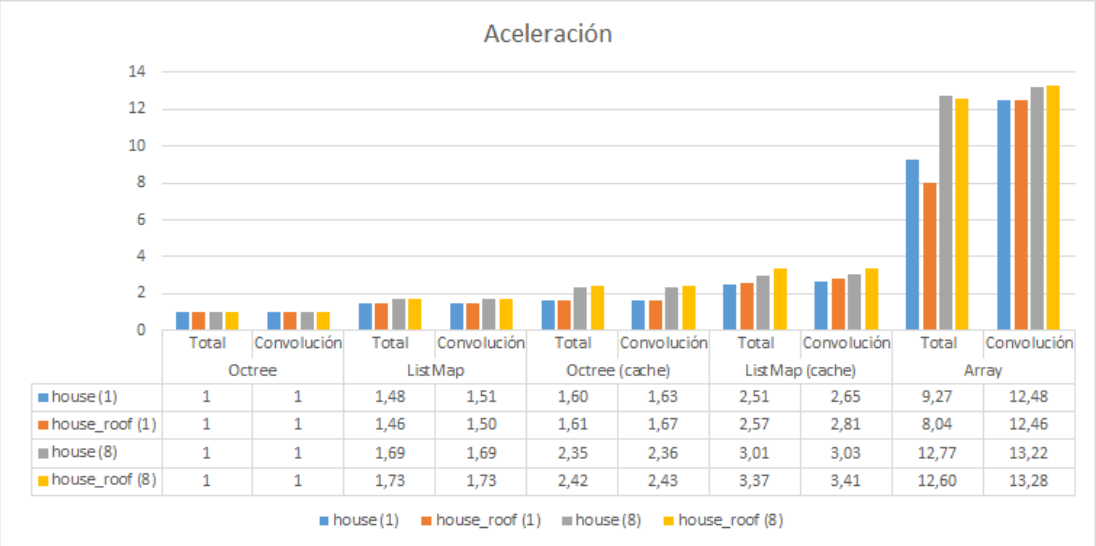
Como se ha mencionado anteriormente, se ha querido experimentar una mejora algorítmica en las versiones secuenciales basadas en árboles octales y mapas, para tratar de equilibrar la contienda respecto a la versión basada en arrays, consistente en la introducción de una caché sustentada por arrays tridimensionales. Como se podrá comprobar en la tabla 2.4 y la figura 2.6 respectivamente, con esta técnica, la mejora de tiempo y aceleración son notables, pero

todavía lejos de la versión íntegramente basada en arrays tridimensionales. Para facilitar la comparación, en la tabla 2.4 se incluyen nuevas filas mostrando la diferencia neta respecto a la versión original. Como referencia, se mantienen los valores de la versión basada en arrays.

		Octree (cache)		ListMap (cache)		Array	
Objeto	Rot.	Total	Convol.	Total	Convol.	Total	Convol.
house	1	12,178	11,579	7,752	7,126	2,101	1,512
		▽7,300	▽7,294	▽5,369	▽5,360	=	=
house_roof	1	7,462	6,867	4,694	4,068	1,498	0,918
		▽4,583	▽4,577	▽3,556	▽3,550	=	=
house	8	89,343	88,728	69,847	69,197	16,472	15,870
		▽121,052	▽121,046	▽54,592	▽54,594	=	=
house_roof	8	54,092	53,478	38,859	38,215	10,396	9,802
		▽76,589	▽76,727	▽36,896	▽36,896	=	=

Cuadro 2.4: Tiempos de ejecución en segundos de los algoritmos secuenciales con caché, y diferencias respecto a las versiones originales

Finalmente, en la figura 2.6 se presentan de forma gráfica y tabulada los datos de **aceleración** de todas las versiones estudiadas.



	Octree		ListMap		Octree (cache)		ListMap (cache)		Array	
	Total	Convolución	Total	Convolución	Total	Convolución	Total	Convolución	Total	Convolución
house (1)	1	1	1,48	1,51	1,60	1,63	2,51	2,65	9,27	12,48
house_roof (1)	1	1	1,46	1,50	1,61	1,67	2,57	2,81	8,04	12,46
house (8)	1	1	1,69	1,69	2,35	2,36	3,01	3,03	12,77	13,22
house_roof (8)	1	1	1,73	1,73	2,42	2,43	3,37	3,41	12,60	13,28

Figura 2.6: Aceleración todos de los algoritmos secuenciales

Capítulo 3

Paralelización en memoria compartida

En este capítulo se exploran distintas alternativas de paralelización sobre un sistema de memoria compartida que se pueden aplicar sobre el algoritmo desarrollado.

Un sistema de memoria compartida se caracteriza porque todos los nodos de computación, generalmente hilos, acceden a una memoria común, a través de la cual se produce la comunicación entre ellos. Este tipo de sistemas escala muy bien con un gran número de nodos y facilita el modelo de programación, pero exigen al desarrollador prestar atención a la consistencia y tomar medidas contra accesos simultáneos a los datos compartidos.

Para este trabajo, las paralelizaciones se realizan mediante la API OpenMP [22][23]. Las operaciones que se han acelerado con OpenMP suponen algo más del 96% del tiempo de ejecución del programa. Son dos secciones, la principal evalúa la convolución propiamente dicha, y la secundaria vuelca la disimilitud calculada sobre la estructura de datos original. Como el tiempo de ejecución de la segunda sección es muy poco significativo, los análisis de ambas operaciones se presentarán de forma conjunta como «tiempo de convolución». Así pues, se presentarán resultados de rendimiento para esta sección conjunta y también para el programa completo, de forma que en este caso se contemple la parte estática de carga de la nube en memoria y preparación de los datos.

Aunque se ha hecho el esfuerzo de paralelizar todas las versiones construidas, este capítulo se centrará en las versiones del algoritmo basadas en estructuras de datos tipo array, con las que se parte de un rendimiento muy superior, tal y como se ha visto en el capítulo anterior.

3.1 Descripción técnica

Como se introducía en la sección dedicada a las estructuras de datos 2.2, la estrategia de paralelización que se revelaba evidente desde el principio era la de dividir el problema inicial en partes de forma que fuese posible disgregar el cómputo entre varios nodos.

En el algoritmo 4 propuesto quedan patentes las estructuras iterativas necesarias para resolver la convolución que también evidencian una primera forma natural de dividir el problema. El análisis de estos bucles revela que son adecuados para ser paralelizados mediante OpenMP, ya que están debidamente estructurados, tienen un número de iteraciones computable y no existen dependencias entre las iteraciones. Así pues, poniendo el foco en la sección de convolución, es posible definir aquí una primera región paralela de la siguiente forma: *#pragma omp parallel for*. En la declaración se incluye la cláusula *private* con la lista de variables que necesitan acceso de escritura en el interior de la sección, de modo que cada hilo obtenga copias privadas. Para concretar el resto de las cláusulas aplicables a este *pragma*, es necesario avanzar un poco más en el análisis de la sección y abordar las dificultades que encierra, ya que la configuración óptima del bucle variará en función de las soluciones aplicadas.

Ya en el pseudocódigo propuesto se identificaba, en el interior de este bucle de convolución, una **sección crítica** en la que se asigna el valor de disimilitud calculado a cada bloque de la vecindad sobre la que se está iterando. Estas posiciones de memoria sufren accesos concurrentes debido a que las vecindades se solapan unas con otras durante las iteraciones.

En una primera versión ingenua se aplica *#pragma omp critical* en la sección que necesita acceso exclusivo mientras compara si el nuevo valor es menor que el actual y escribe este nuevo valor en caso de ser necesario. Con esta disposición ya es posible obtener resultados consistentes con la versión secuencial. Sin embargo, se observa una pérdida de rendimiento y eficiencia muy significativa en comparación con una tentativa sin sección crítica -aunque ésta evidentemente no produzca resultados consistentes, se puede tomar como referencia del mejor rendimiento posible-. Para el caso de prueba de referencia, y utilizando 24 núcleos, el tiempo de ejecución sin una sección crítica es un 99% inferior, por lo que es indudable que hay un margen de mejora muy importante. El foco se pone entonces en la eliminación de esa sección crítica, en la que, para cada elemento de la malla, se está almacenando el valor mínimo de disimilitud. Esta operación es manifiestamente compatible con una reducción al mínimo.

La optimización pasa entonces por efectuar una nueva iteración de desarrollo para adaptar el programa de forma que se disponga de una estructura de datos sobre la cual efectuar dicha reducción directamente. Se construye a tal efecto un array tridimensional de salida que almacene directamente los valores de disimilitud y se aplica una reducción al mínimo sobre dicho array. El algoritmo sigue utilizando para lectura la estructura de datos dispuesta originalmente. Se muestran los detalles generales de esta implementación en el listado 3.1.

Listing 3.1: Bucles de convolución paralelizados

```
// Definición e inicialización del array de disimilitud
double d[dim.x][dim.y][dim.z];
for (int x = 0; x < dim.x; x++) {
    for (int y = 0; y < dim.y; y++) {
        for (int z = 0; z < dim.z; z++) {
            d[x][y][z] = DBL_MAX;
        }
    }
}
[...]
// Convolución y reducción sobre el array de disimilitud
#pragma omp parallel for [...] \
    private(x, y, z, [...]) \
    reduction(min:d[:dim.x][:dim.y][:dim.z])
for (x = 0; x < dim.x; x++) {
    for (y = 0; y < dim.y; y++) {
        for (z = 0; z < dim.z; z++) {
            [...]
        }
    }
}
```

Con esta refactorización se logra una reducción del 99% en el tiempo de ejecución, equiparando así su rendimiento al del código sin sección crítica. En este punto, resulta necesario aumentar el tamaño del problema para ver diferencias significativas, dado que la ejecución del caso de prueba inicial se completa ahora en pocos milisegundos. No obstante, este caso de prueba inicial ha sido de gran utilidad para agilizar el proceso de desarrollo hasta el momento.

Con el problema de la sección crítica resuelto, a continuación se realizan algunos experimentos para encontrar de forma empírica el mejor comportamiento del *#pragma omp for*. Principalmente se hacen ensayos con las cláusulas *collapse* y *schedule*.

En el caso de la cláusula *collapse*, se contrasta la posibilidad de fusionar los tres bucles (*collapse(3)*) y el comportamiento por defecto, en el que se paraleliza únicamente el bucle externo. La diferencia de rendimiento se hace patente muy lentamente a medida que aumenta el número de iteraciones, es decir, el tamaño del problema, ofreciendo mejor rendimiento la versión sin fusionar. Se entiende que esta opción presenta una menor sobrecarga en las iteraciones y/o se ve beneficiada ligeramente por un mejor acceso a los datos.

En el caso de la cláusula *schedule*, la estrategia que ofrece el mejor rendimiento es la planificación dinámica *schedule(dynamic)*, probablemente debido a su flexibilidad y adaptabilidad a la hora de distribuir el trabajo entre los hilos, si bien, las diferencias con el resto de planifi-

caciones analizadas no eran marcadamente grandes, y también es necesario irse a problemas grandes para empezar a ver diferencias.

La segunda sección optimizada, el bucle de copia de los resultados de disimilitud obtenidos sobre la estructura de datos original, ya no presenta dificultades de tipo alguno y permite ser paralelizada con *#pragma omp parallel for [...] private(x, y, z, [...])*. Se ejecuta una batería de pruebas similar al caso anterior para tratar de afinar el comportamiento, pero la variabilidad es tan poco significativa, que se opta por una configuración idéntica al bucle anterior.

3.1.1 Medidas de rendimiento

Como se comentaba en la sección anterior, dado el rendimiento superior de esta versión paralelizada, se requiere un problema de mayor envergadura que permita obtener diferencias más notables entre las distintas configuraciones de ejecución. Por esta razón, se comienza a utilizar la escena *5080_54400_big_segment* y un tamaño de bloque 0,5 como caso base. Además, lo que interesa observar en esta sección es el comportamiento de la versión paralelizada a través de las métricas de aceleración y eficiencia evaluadas tras las mediciones realizadas.

En las tablas 3.1 y 3.2 se recogen respectivamente los tiempos de ejecución total y de convolución en segundos que se observan en función del nivel de paralelismo (número de hilos o cores utilizados) para los dos objetos de búsqueda (*kernel*) que se han venido utilizando.

En cuanto a los valores de aceleración y eficiencia presentados en las figuras 3.1 y 3.2, conviene aclarar que, para cada caso de prueba -caracterizado por la escena, el objeto buscado, el tamaño de bloque y el número de rotaciones-, su cálculo se efectuará tomando como referencia la ejecución de la versión actual en un único hilo, y no la ejecución de la versión secuencial presentada en el capítulo anterior.

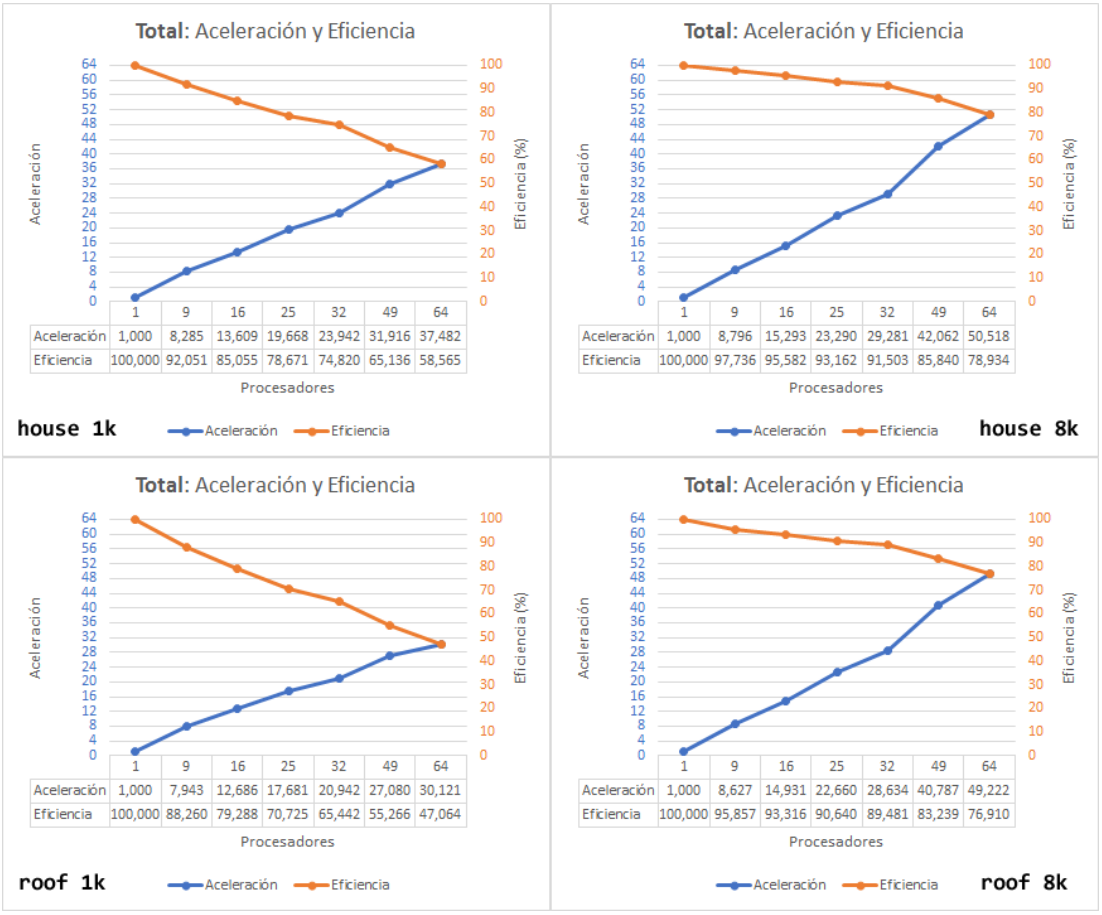
Se comienza por la tabla 3.1 que recoge los **tiempos de ejecución total** observados para los nuevos casos de prueba.

		Cores						
Obj.	Rot.	1	9	16	25	32	49	64
house	1	293,630	35,442	21,576	14,929	12,263	9,199	7,833
house_roof	1	181,622	22,864	14,316	10,272	8,672	6,706	6,029
house	8	3204,718	364,327	209,553	137,597	109,446	76,191	63,437
house_roof	8	1994,459	231,185	133,582	88,017	69,653	48,899	40,519

Cuadro 3.1: Tiempo total en segundos para *5080_54400_big_segment* y tamaño de bloque 0,5

En la figura 3.1, que muestra las gráficas de aceleración y eficiencia evaluadas con los datos de la tabla 3.1, se podrá comprobar que la tendencia de aceleración es lineal y que tanto los valores de aceleración como los de eficiencia obtenidos serán mejores o peores en función

del número de rotaciones configuradas. Cuanto menor sea el número de rotaciones, el tiempo de ejecución de la parte estática será proporcionalmente más significativo en el tiempo de ejecución total del programa, ya que el tiempo de cálculo de la convolución será menor, y por tanto, se alcanzará menor aceleración y eficiencia.



house 1k	1	9	16	25	32	49	64
Aceleración	1,000	8,285	13,609	19,668	23,942	31,916	37,482
Eficiencia	100,000	92,051	85,055	78,671	74,820	65,136	58,565

house 8k	1	9	16	25	32	49	64
Aceleración	1,000	8,796	15,293	23,290	29,281	42,062	50,518
Eficiencia	100,000	97,736	95,582	93,162	91,503	85,840	78,934

roof 1k	1	9	16	25	32	49	64
Aceleración	1,000	7,943	12,686	17,681	20,942	27,080	30,121
Eficiencia	100,000	88,260	79,288	70,725	65,442	55,266	47,064

roof 8k	1	9	16	25	32	49	64
Aceleración	1,000	8,627	14,931	22,660	28,634	40,787	49,222
Eficiencia	100,000	95,857	93,316	90,640	89,481	83,239	76,910

Figura 3.1: Aceleración y eficiencia del programa completo

A modo de resumen, se ha podido ver en las gráficas de la figura 3.1 que, para el máximo nivel de paralelismo con 64 hilos, la aceleración alcanzada se sitúa entre 30 y 37 puntos en el caso no efectuar rotaciones, y alrededor de 50 puntos para el caso de efectuar 8 rotaciones. Por su parte, la eficiencia se sitúa entre el 47% y el 58% y entre el 77% y el 79% respectivamente. Son unos números razonablemente buenos tratándose de medidas de tiempo total del programa.

Por otro lado, es interesante observar que la proporción de mejora del tiempo de ejecución al utilizar el objeto *5080_55500_house_roof* como *kernel* en lugar del objeto *5080_55500_house* también se degrada paulatinamente a medida que aumenta el número de núcleos empleado, aunque esta degradación es mucho menor si sólo se considera la sección de convolución.

A continuación, en la tabla 3.2 se recogen los **tiempos de convolución** en segundos.

Obj.	Rot.	Cores 1	9	16	25	32	49	64
house	1	291,036	32,702	18,881	12,241	9,606	6,494	5,132
house_roof	1	178,916	20,139	11,643	7,548	5,966	3,988	3,305
house	8	3201,643	361,626	206,772	134,792	106,695	73,434	60,604
house_roof	8	1991,799	228,436	130,777	85,264	66,911	46,092	37,721

Cuadro 3.2: Tiempo de convolución en segundos para *5080_54400_big_segment* y tamaño de bloque *0,5*

En la figura 3.2 se presentan las gráficas de aceleración y eficiencia. Para el máximo nivel de paralelismo comprobado, la aceleración se sitúa entre 53 y 57 puntos y la eficiencia entre el 83% y el 88%, lo que indica una baja degradación y un notable éxito de la paralelización.

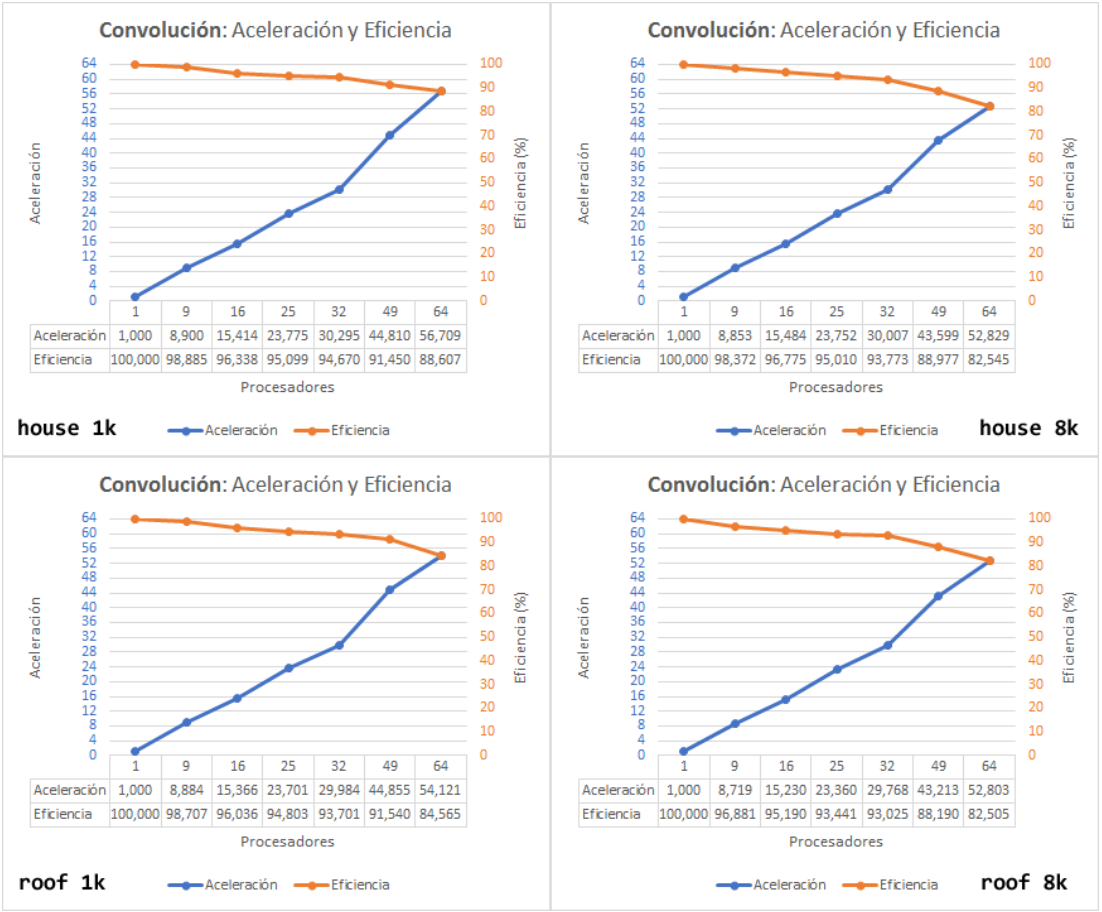


Figura 3.2: Aceleración y eficiencia del algoritmo de convolución

De forma complementaria, en la figura 3.3 se muestran las salidas de la convolución para los casos de prueba descritos. Al aumentar la precisión y el tamaño de la escena, se puede confirmar definitivamente que el objeto *5080_55500_house_roof* produce mejores resultados.

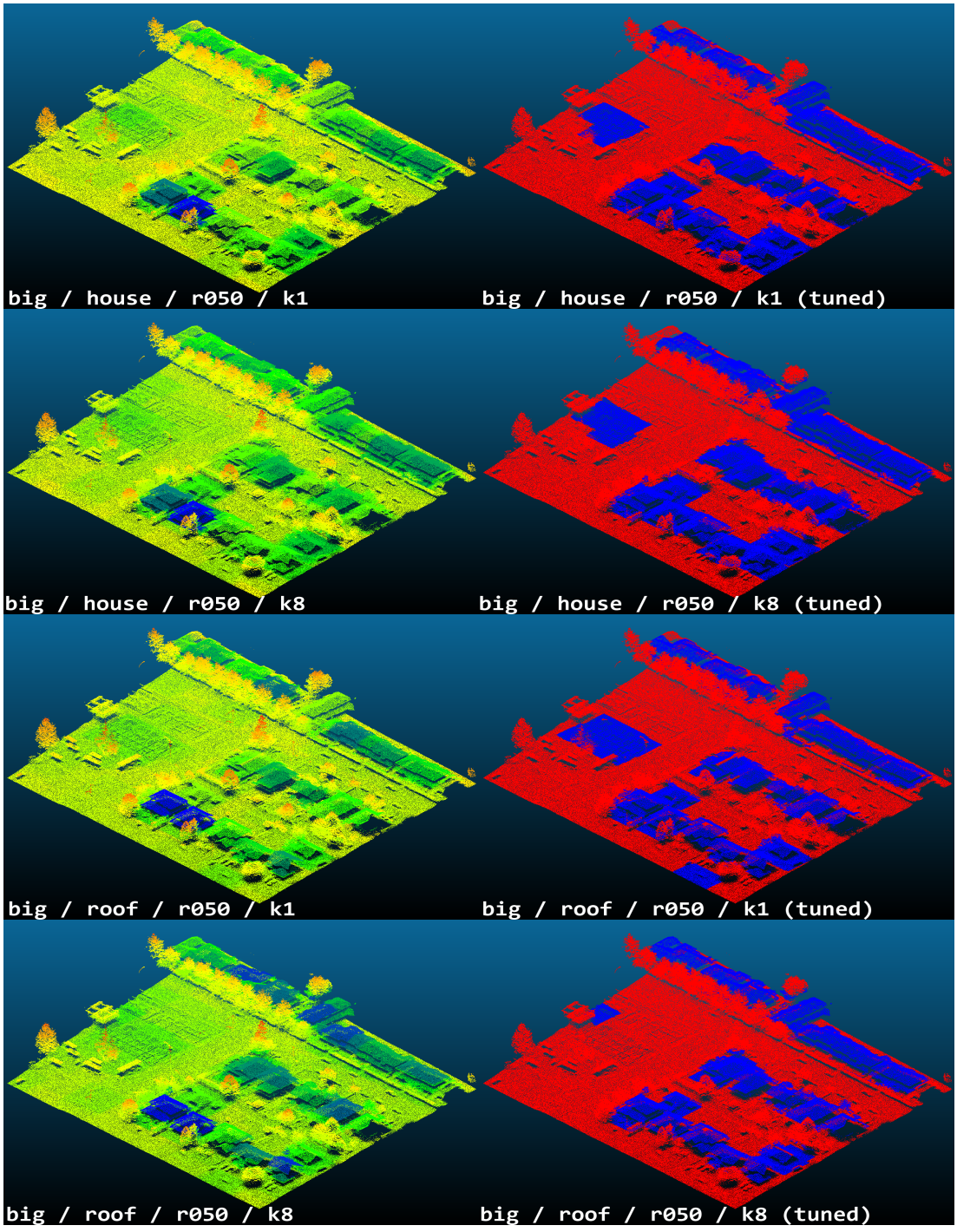


Figura 3.3: Salida para escena grande con tamaño de bloque 0,5

A lo largo de todo el documento se viene dando importancia a la cuestión de la representación de la información en memoria y al consumo de recursos. Ahora que es posible realizar convoluciones sobre escenas grandes, es importante observar qué ocurre con el consumo de memoria. Se han realizado mediciones para la misma escena, *5080_54400_big_segment*, objeto *5080_54400_house_roof*, y tamaños de bloque 0,5 y 0,25. Se muestra a continuación el uso de memoria residente, en GiB, que utiliza el proceso en las distintas situaciones.

	Cores						
Tamaño de bloque	1	9	16	25	32	49	64
0,5	0,3	0,7	1,0	1,5	1,8	2,6	3,4
0,25	1,6	4,6	7,2	10,6	13,2	19,5	25,1

Cuadro 3.3: Consumo de memoria en GiB de la ejecución del algoritmo basado en arrays

Lo que se puede observar es un consumo de memoria que se dispara a medida que aumenta el número de hilos (cores) y la precisión (tamaño de bloque) exigida. Se puede comprobar gráficamente en 3.4.

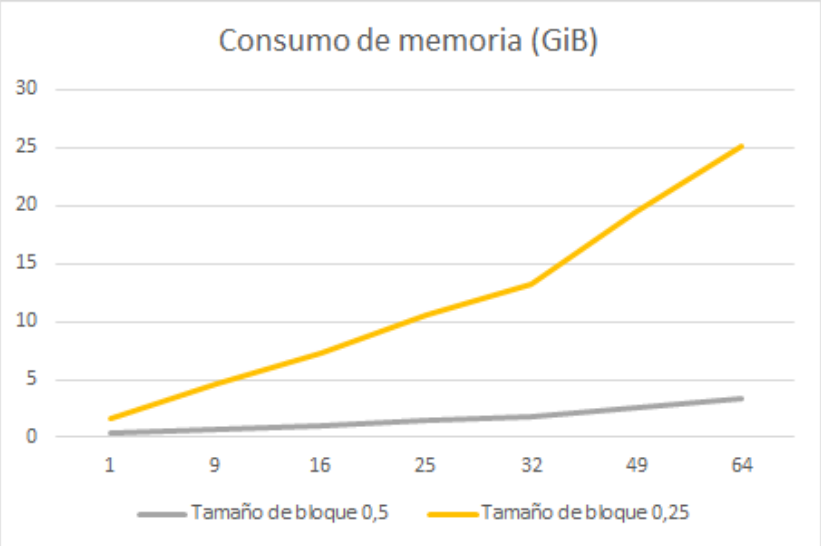


Figura 3.4: Consumo de memoria de la ejecución paralela del algoritmo basado en arrays

En cuanto al número de hilos, lo que ocurre está directamente relacionado con la estrategia de paralelización basada en la reducción al mínimo sobre un array. Y es que para cada hilo se crea una copia propia del array de disimilitud en el que cada uno de ellos escribe de forma exclusiva sus valores, hasta el paso final de fusión. Por tanto, el tamaño de memoria necesario es proporcional al nivel de paralelismo.

Como se ha indicado anteriormente, el tamaño de bloque influye directamente sobre el tamaño de este array de disimilitud, a menor tamaño de bloque, mayor tamaño del array, ya que se segmenta la nube de puntos con mayor granularidad. Debido al ámbito de declaración del array y la estrategia de reducción, las copias de este array se crean en la memoria de pila creada para cada uno de los hilos. Sucede entonces que, cuando el array necesita demasiado espacio, puede desbordar pila, y en tales situaciones es necesario ajustar la variable *OMP_STACKSIZE* que controla el tamaño de la pila creada para cada hilo, de modo que sea posible llevar la ejecución a buen término. Como ejemplo, para las ejecuciones anteriores se ha necesitado establecer la variable a *52M* para tamaño de bloque *0,5* y *400M* para tamaño de bloque *0,25*. Como ayuda, el programa muestra el tamaño necesario para alojar una instancia de este array, aunque siempre es necesario asignar algo más para otros usos.

Volviendo a los resultados mostrados anteriormente, y como resumen, el programa ofrece una velocidad razonable, una aceleración y eficiencia buenas, pero el consumo de memoria parece un tanto desmesurado, sobre todo si se compara con el tamaño que ocupa la información en disco: 34 MiB la escena *5080_54400_big_segment*. Y es por esta razón por la que se realiza una iteración más de desarrollo.

3.1.2 Optimización de consumo de recursos

A la vista de los resultados comentados en la sección anterior, se hace patente la necesidad de rebajar las altas demandas de memoria del programa para permitir manejar escenarios mucho más grandes con menos recursos. Teniendo ya identificada la causa del alto consumo de memoria, el nuevo objetivo sería lograr la paralelización del programa sin utilizar la reducción al mínimo estándar que provoca el problema.

Como punto de partida, el programa debe igualmente repartir el trabajo entre el número de hilos disponible. Por tanto, conociendo este número y el tamaño de la escena, es posible hacer una partición equitativa de la escena. Para realizar esta partición, la estrategia utilizada descompone el número de hilos en los dos factores más próximos que es posible encontrar. Los factores encontrados se asignan a los ejes X e Y, significando el número de divisiones realizadas en cada dimensión, mientras que el eje Z no se divide. Por ejemplo, para 24 hilos la descomposición obtenida sería 4×6, dividiendo en 4 partes el eje X, en 6 el eje Y y 1 en el eje Z.

Teniendo acotada la parte (ventana) del problema en que se centrará cada hilo, para cada uno de ellos, también es necesario construir un array de disimilitud del mismo tamaño que dicha partición. Por tanto, ahora no existiría una copia completa del array de disimilitud por cada hilo, sólo una serie de copias parciales que, vueltas a reunir, ocuparían el tamaño de una copia. En realidad, algo más, porque las particiones se necesitan solapar unas con otras en sus límites, ya que así es como trabaja la convolución.

Gráficamente, podría representarse la situación como se muestra en la figura 3.5. Los bloques sombreados en amarillo corresponderían exactamente a la primera partición, mientras que los sombreados en rojo estarían fuera de sus límites, pero se deben contemplar debido al tamaño del kernel y el solape que provoca. En este ejemplo, se representaría la convolución en 4 hilos de una nube que queda descompuesta en 12 x 8 x 6 bloques debido a su tamaño y el tamaño de bloque configurado, mientras que el kernel queda descompuesto en 3×3 debido a su propio tamaño y el tamaño de bloque configurado.

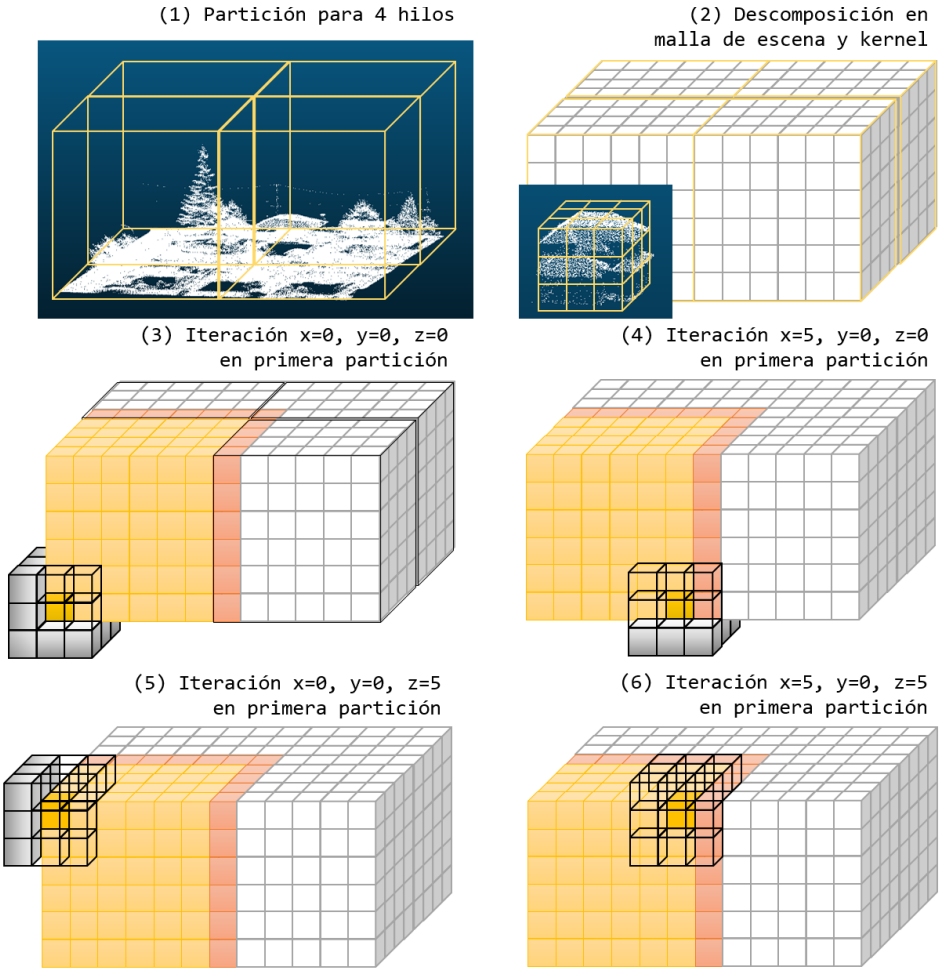


Figura 3.5: Representación gráfica de la convolución y zonas solapadas

Una vez dividido el problema, y una vez que cada hilo ha efectuado su parte de la convolución, ahora es necesario realizar la fusión de los resultados en un único array de disimilitud, imitando la reducción al mínimo original. Este paso final se realiza de forma secuencial ya que, precisamente por los solapes, valores en distintos arrays parciales convergen en la misma posición del array de disimilitud final. Además, ya es un problema computacionalmente

mucho más sencillo.

A pesar de todas las refactorizaciones descritas, en esencia, el algoritmo no cambia, ya que produce los mismos resultados con los mismos parámetros de entrada.

Respecto a la paralelización, ahora se efectúa con una sencilla sección *#pragma omp parallel* en la que cada hilo es consciente, en función de su identificador, de la sección de la nube que le corresponde convolucionar, que se sitúa en memoria compartida, y de su array de desimilitud de salida asociado. En esta sección paralela únicamente se configuran las variables privadas necesarias para evitar sobrecargas de declaración.

3.1.3 Medidas de rendimiento del algoritmo alternativo

En esta sección, se muestran los resultados de enfrentar esta nueva versión optimizada a los mismos problemas que la versión de partida.

En la tabla 3.4 se muestran los tiempos de ejecución total de la nueva versión del programa. Como lo ideal es contrastar los resultados con los de la anterior versión, se incluyen además nuevas filas que muestran el incremento (△) o decremento (▽) del tiempo recogido para los mismos casos en la tabla 3.1.

		Cores						
Obj.	Rot.	1	9	16	25	32	49	64
house	1	292,612	39,164	25,067	16,546	14,441	11,004	9,281
		▽1,017	△3,721	△3,491	△1,617	△2,177	△1,804	△1,447
house_roof	1	182,605	24,876	16,470	11,297	9,890	7,862	6,700
		△0,983	△2,011	△2,154	△1,025	△1,217	△1,155	△0,670
house	8	2977,019	371,65	230,726	146,068	119,941	86,549	71,87
		▽227,698	△7,323	△21,173	△8,470	△10,494	△10,357	△8,433
house_roof	8	1808,333	225,132	139,503	88,116	73,429	53,218	43,694
		▽186,126	▽6,052	△5,920	△0,098	△3,775	△4,319	△3,175

Cuadro 3.4: Tiempo total en segundos para *5080_54400_big_segment* y tamaño de bloque 0,5 para la versión de memoria reducida, y diferencias respecto a la original

La que se puede observar es que, en términos de velocidad, la nueva versión está muy cerca comparativamente de la versión anterior, oscilando la diferencia entre un 7% más rápida y un 13% más lenta, si bien, en términos generales, el balance indica una ligera pérdida de rendimiento. La inclusión de la nueva sección secuencial en la que se realiza la reducción induce una pequeña pérdida de eficiencia del programa. Igual que en el caso de partida, y por las mismas razones, el programa muestra peor eficiencia cuando no se efectúan rotaciones, entre el 42% y el 49% en contraste con el 64% que se obtiene efectuando 8 rotaciones.

En la figura 3.6 se muestran las gráficas de aceleración y eficiencia evaluadas según los datos recogidos en la tabla 3.4.

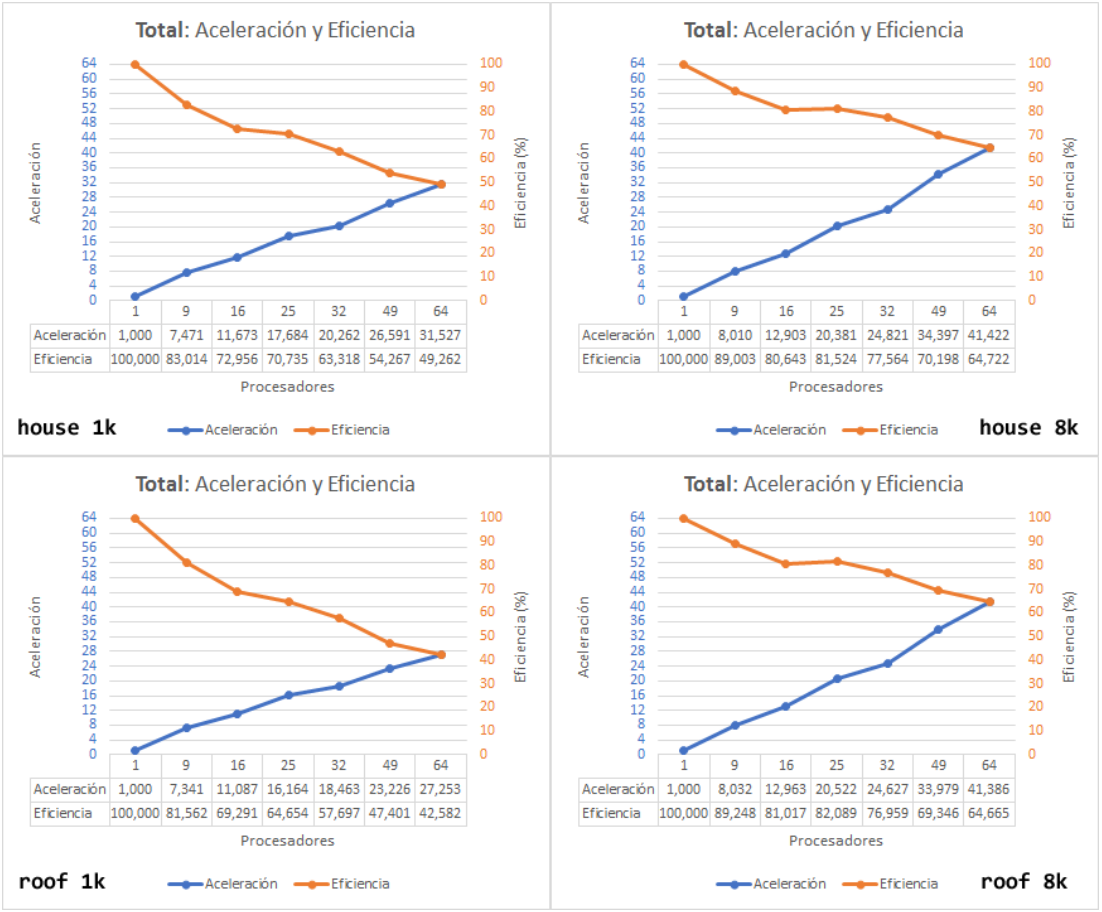


Figura 3.6: Aceleración y eficiencia del programa completo con la versión optimizada para un consumo de memoria reducido

En la tabla 3.5 se recogen los datos relativos al **tiempo de convolución**, incluyendo también las diferencias respecto a los datos recogidos en la tabla 3.2 correspondiente a la versión de partida. En los datos recogidos se podrá notar que las diferencias de tiempo recogidas son prácticamente las mismas que en la tabla de tiempo de ejecución total, lo que se explica porque los cambios en el programa afectan íntegramente al algoritmo de convolución. Además de los reajustes para manejar las ventanas, la ejecución de la nueva sección secuencial que efectúa la reducción final se ha considerado como tiempo de convolución, para ser justos en la comparación con el algoritmo anterior. Este hecho explica la pérdida de eficiencia que se podrá observar en la figura 3.7, en la que se muestran las correspondientes gráficas de aceleración y eficiencia confeccionadas con los datos de la tabla 3.5. La eficiencia de la convolución para el máximo nivel de paralelismo se sitúa entre un 67% y un 71% todavía aceptable.

Obj.	Rot.	Cores 1	9	16	25	32	49	64
house	1	289,941	36,454	22,316	13,681	11,626	8,126	6,491
		▽1,095	△3,752	△3,435	△1,439	△2,019	△1,632	△1,359
house_roof	1	179,913	22,115	13,662	8,468	7,123	5,064	3,953
		△0,997	△1,975	△2,019	△0,919	△1,156	△1,075	△0,647
house	8	2974,259	368,605	227,904	142,886	117,137	83,653	69,07
		▽227,384	△6,978	△21,132	△8,094	△10,442	△10,219	△8,465
house_roof	8	1805,551	222,29	136,676	85,262	70,542	50,359	40,83
		▽186,247	▽6,145	△5,898	▽0,001	△3,631	△4,267	△3,109

Cuadro 3.5: Tiempo de convolución en segundos para *5080_54400_big_segment* y tamaño de bloque 0,5 para la versión de memoria reducida, y diferencias respecto a la original

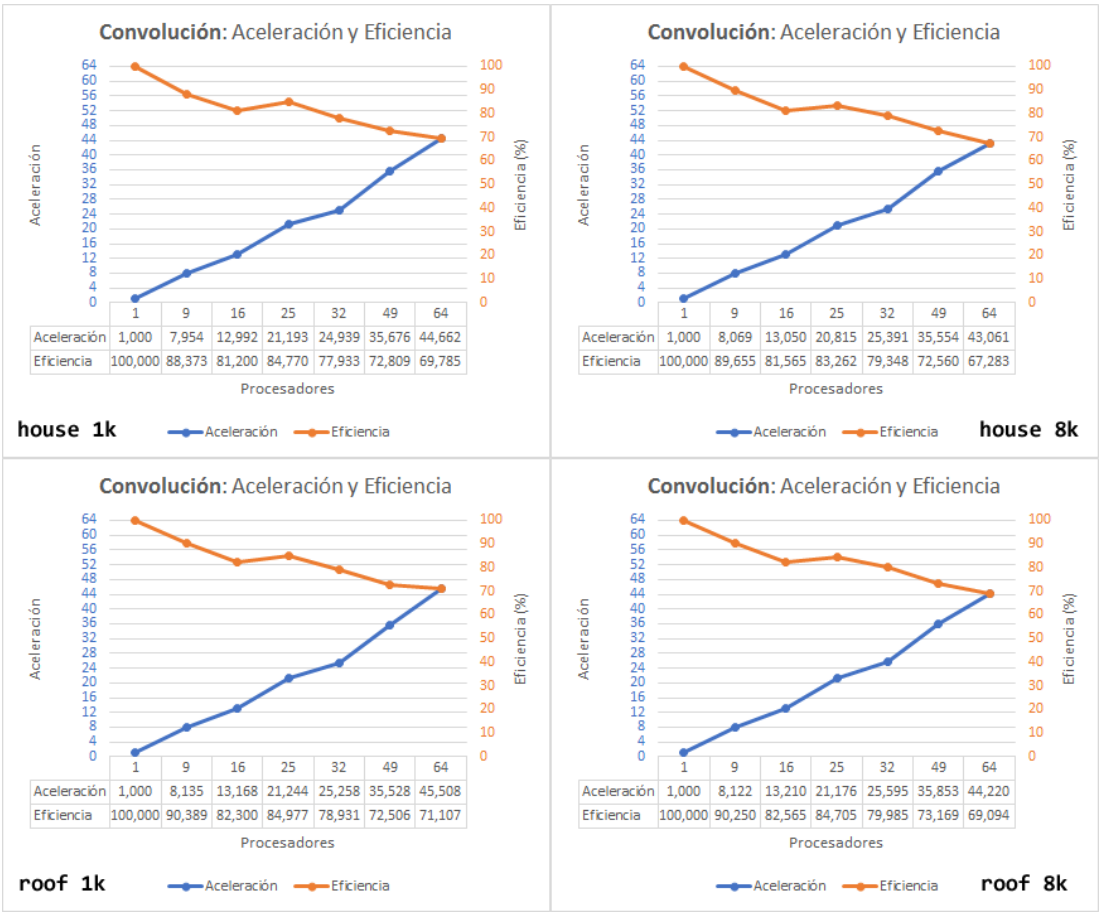


Figura 3.7: Aceleración y eficiencia del algoritmo de convolución con la versión optimizada para un consumo de memoria reducido

En cuanto a la memoria, razón de ser de esta nueva versión, estos son ahora los consumos observados.

	Cores						
Tamaño de bloque	1	9	16	25	32	49	64
0,5	0,28	0,32	0,34	0,36	0,38	0,41	0,44
0,25	1,50	1,70	1,90	2,10	2,20	2,40	2,70

Cuadro 3.6: Consumo de memoria en GiB de la ejecución paralela del algoritmo basado en arrays y optimizado

Los datos confirman la validez de la estrategia y la drástica reducción del consumo de memoria respecto a lo recogido en la tabla 3.3. En la gráfica 3.8 se puede apreciar visualmente la diferencia de consumo para los dos casos evaluados. El consumo de memoria se incrementa ahora de forma lineal y lentamente, confiriendo al programa una mayor usabilidad para problemas reales.

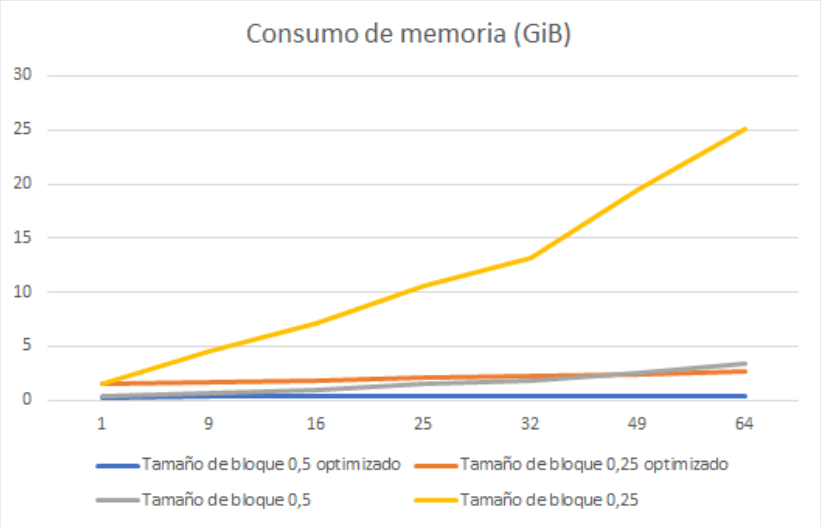


Figura 3.8: Comparativa de consumo de memoria entre la versión anterior y la optimizada

3.2 Presentación de resultados finales

Para finalizar, en la figura 3.9 se muestran los resultados de convolucionar la escena original con los objetos de búsqueda que se han venido utilizando como referencia en este trabajo, con tamaño de bloque 0,5 y 8 rotaciones. Ejecuciones realizadas con la última versión del algoritmo y que se han completado en 21m 51s en el caso del recorte *5080_54400_house* y en 11m 35s en el caso de *5080_54400_house_roof*.

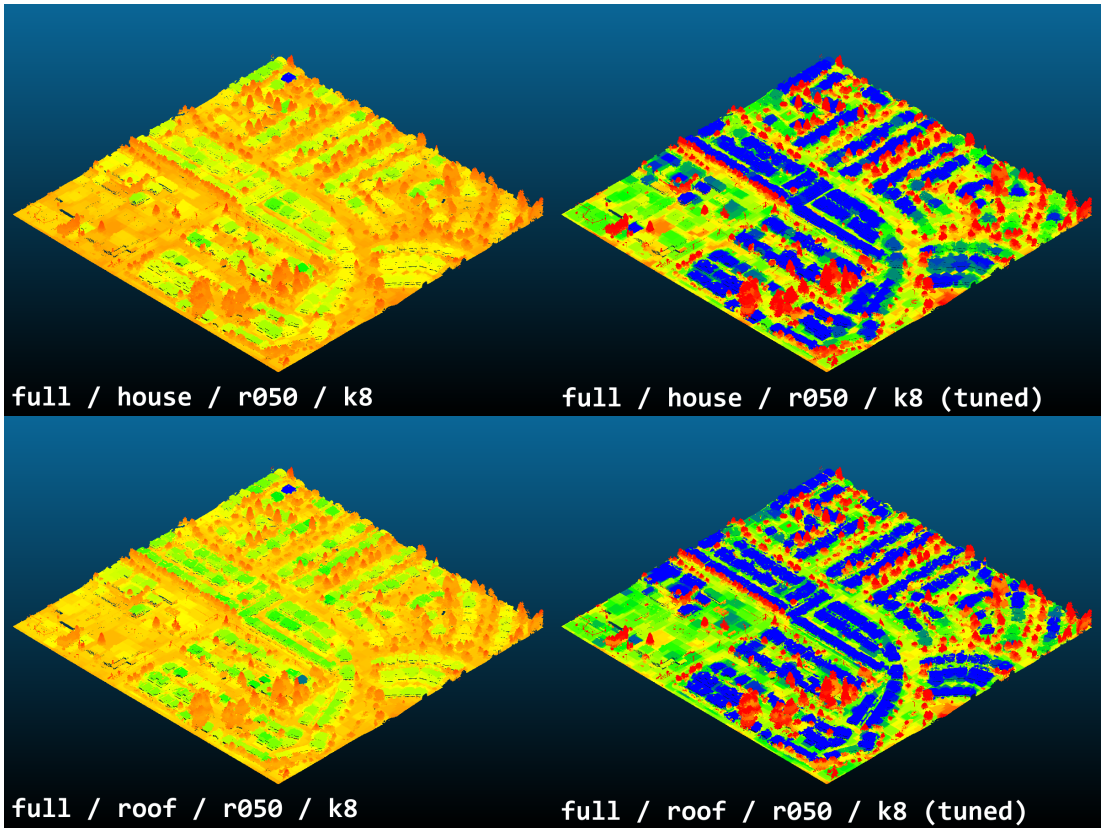


Figura 3.9: Convolución de la escena completa con tamaño de bloque *0,5*

Tal y como se ha venido indicando, y como última conclusión a la vista de los resultados, se puede afirmar que la precisión percibida como resultado de la convolución depende decisivamente del objeto de búsqueda utilizado como *kernel*.

Capítulo 4

Conclusiones

Partiendo del objetivo de lograr la identificación de edificios en las escenas seleccionadas como casos de partida, a lo largo de este trabajo, se han expuesto las pruebas visuales que confirman su consecución.

Desde la primera versión secuencial que consigue resultados aceptables como salida de la convolución, se alcanzan aceleraciones de hasta 652 puntos suponiendo una reducción de tiempo de ejecución del 99,85%. Esta aceleración se consigue mediante el uso de prácticas de programación para un rendimiento eficiente, la introducción de mejores representaciones de datos y, por supuesto, la paralelización de los algoritmos desarrollados. Como se ha demostrado, la estructura de datos que mejor se adapta al problema de las convoluciones es, sin duda, el array tridimensional. La rapidez con la que se consigue direccionar en memoria y la contigüidad de datos son sus dos bazas determinantes.

Respecto al trabajo de paralelización realizado en la segunda parte de este trabajo, OpenMP resulta ser una herramienta tremendamente eficaz para la paralelización de programas en memoria compartida, ya que permite con muy poco esfuerzo actuar sobre algoritmos diseñados de forma general para obtener versiones capaces de rendir con alta eficiencia. En esta etapa, se ha conseguido mediante la paralelización del programa en 64 hilos (núcleos) una reducción de tiempo de ejecución de la convolución de alrededor del 98% respecto al tiempo de ejecución secuencial, consiguiendo aceleraciones de más de 56 puntos y eficiencia de casi 89% en el mejor de los casos.

Se han obtenido resultados razonablemente buenos con tamaños de bloque grandes y sin introducir rotaciones del objeto buscado, debido a la coincidencia en la orientación del objeto con la escena, y a la relativa homogeneidad de la escena casualmente elegida. Bajo estas condiciones la salida se obtiene con gran velocidad y las medidas indican muy alta eficiencia. No obstante, se observa un buen comportamiento general en todo tipo de zonas a medida que se aumenta la precisión, disminuyendo el tamaño de bloque, y se aumenta el número de rotaciones aunque, lógicamente, la velocidad y la eficiencia medidas disminuyen en consecuencia.

Como se ha podido ver, la estructura elegida para la representación de la nube de puntos, un alto grado de paralelismo y el uso de reducciones OpenMP en la convolución inducen un alto consumo de memoria. Para lidiar con este inconveniente, ha sido necesario recurrir a una estrategia más ”artesanal” en la división del trabajo, pivotando el algoritmo hacia una versión ad-hoc pensada específicamente para su paralelización. Con esta versión se ha conseguido un rendimiento muy ligeramente inferior y con un consumo de memoria drásticamente menor, en el que se pasa de una progresión de consumo geométrica a una progresión aritmética. Como ejemplo, en el caso más significativo de los estudiados se consigue reducir de 25,1 GiB a 2,7 GiB.

Existen algunas cuestiones que ya no se abordan en este trabajo por falta de tiempo material. Una podría ser la paralelización con MPI. Puesto que en esta última versión ya está construida la lógica de partición del problema y la de fusión de los resultados, se podría considerar que está un paso más cerca para ser paralelizada en sistemas de memoria distribuida. Sería posible construir una versión que distribuya el trabajo entre nodos físicos y en cada uno de ellos recurrir de nuevo a OpenMP para maximizar el rendimiento y dotar al programa de capacidad para procesar escenas cada vez mayores. Otra cuestión muy interesante hubiese sido el desarrollo de un método capaz de determinar la precisión del algoritmo comparando los resultados contra la clasificación de puntos que proporciona el propio *dataset* de DALES. La dificultad aquí radicaría en la determinación de los umbrales de disimilitud que se debieran estipular para considerar la coincidencia con el objeto buscado, ya que los valores con los que visualmente sí podemos interpretar una identificación positiva, varían dependiendo de varios factores, como la escena, el tamaño de bloque y la idoneidad del objeto de búsqueda utilizado.

Apéndice A

Glosario de acrónimos

HPC *High Performance Computing.*

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

DALES *Dayton Annotated Laser Earth Scan.*

CNN *Convolutional Neural Network.*

ANN *Artificial Neural Network.*

FLOPS *Floating Point Operations Per Second.*

RAM *Random Access Memory.*

SSD *Solid State Disk.*

NVMe *Non Volatile Memory Express.*

API *Application Programming Interface.*

OMP *OpenMP.*

MPI *Message Passing Interface.*

Apéndice B

Glosario de términos

Nube de puntos 3D Una nube de puntos es un conjunto de datos tridimensionales que representan la posición de puntos en el espacio tridimensional.

Convolución La convolución es una operación matemática que combina dos funciones para producir una tercera función que representa cómo una de las dos funciones modifica la forma de la otra. En el contexto del procesamiento de señales y la visión por computadora, la convolución se utiliza para aplicar un filtro o una máscara a una señal o una imagen. Esto implica superponer la máscara sobre la señal o la imagen y calcular el promedio ponderado de los valores de píxel o de muestra en la región de la máscara para generar una nueva señal o imagen filtrada. La convolución es una operación fundamental en muchas aplicaciones de procesamiento de señales, como el filtrado, la detección de características y el procesamiento de imágenes. [24]

Reconocimiento de objetos El reconocimiento objetos se refiere al proceso mediante el cual un sistema computacional identifica y clasifica objetos dentro de una imagen o escena.

Red neuronal artificial En el ámbito del aprendizaje automático, una red neuronal artificial es un modelo inspirado en la estructura y función de las redes neuronales biológicas del cerebro de los animales. Estas redes están compuestas por unidades llamadas neuronas artificiales que están conectadas entre sí mediante bordes que representan las sinapsis del cerebro. Cada neurona artificial recibe señales de las neuronas conectadas, las procesa y envía una señal a otras neuronas. Las señales se calculan mediante una función de activación y los pesos de las conexiones se ajustan durante el proceso de aprendizaje. Las neuronas se agrupan en capas, y estas capas pueden realizar diferentes transformaciones en las entradas. Las redes neuronales artificiales se utilizan para diversas tareas y pueden aprender de la experiencia y extraer conclusiones a partir de información compleja y aparentemente inconexa.

Red neuronal convolucional Una red neuronal convolucional (CNN) es un tipo regularizado de red neuronal retro alimentada que aprende a extraer atributos por sí misma mediante la optimización de filtros (o kernel). Este tipo de redes están especialmente diseñadas para procesar datos con una estructura de cuadrícula, como imágenes o, en este caso, nubes de puntos 3D.

Vóxel En informática gráfica 3D, un vóxel es una imagen de una región espacial tridimensional limitada por tamaños dados, que tiene sus propias coordenadas de punto nodal en un sistema de coordenadas aceptado, su propia forma, su propio parámetro de estado que indica su pertenencia a algún objeto modelado, y tiene propiedades de región modelada.

Aceleración La aceleración es una medida del cambio en el tiempo de ejecución de un proceso después de aplicar una optimización específica. Se calcula como la relación entre el tiempo de ejecución antes y después de la optimización.

Eficiencia La eficiencia es una métrica que evalúa la utilización de recursos en un sistema paralelo. Se calcula como la relación entre la aceleración y el número de nodos de procesamiento, lo que proporciona una medida de cuán efectivamente se están utilizando los recursos disponibles en el sistema.

Bibliografía

[1] "Point cloud," Available online at: https://en.wikipedia.org/wiki/Point_cloud (Accessed: 25 May 2024).

[2] "What is LiDAR?" Available online at: https://www.ibm.com/topics/lidar (Accessed: 25 May 2024).

[3] A. Jelalian, *Laser Radar Systems*, ser. Artech House radar library. Artech House, 1992.

[4] "Dayton Annotated Laser Earth Scan (DALES)," Available online at: https://udayton.edu/engineering/research/centers/vision_lab/research/was_data_analysis_and_processing/dale.php (Accessed: 25 May 2024).

[5] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 922–928.

[6] L. P. Tchapmi, C. B. Choy, I. Armeni, J. Gwak, and S. Savarese, "Segcloud: Semantic segmentation of 3d point clouds," in *2017 International Conference on 3D Vision (3DV)*. IEEE, 2017, pp. 537–547.

[7] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1912–1920.

[8] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, "Pointcnn: Convolution on x-transformed points," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 820–830.

[9] G. Riegler, A. Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 3577–3586.

[10] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 5, p. 146, 2019.

[11] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 652–660.

[12] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5099–5108.

[13] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6526–6534.

[14] B. Yang, W. Luo, and R. Urtasun, "Pixor: Real-time 3d object detection from point clouds," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 7652–7660.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.*, 6th ed. Morgan Kaufmann Publishers, 2019.

[16] "Acerca del CESGA," Available online at: https://www.cesga.es/cesga/el-cesga/ (Accessed: 25 May 2024).

[17] "Finisterrae III User Guide: Overview," Available online at: https://cesga-docs.gitlab.io/ft3-user-guide/overview.html (Accessed: 25 May 2024).

[18] C. H. Edwards and D. E. Penney, *Ecuaciones diferenciales y problemas con valores en la frontera. Cómputo y modelado.*, 4th ed. México: Pearson Educación, 2009.

[19] S. B. Damelin and W. M. Jr., *The Mathematics of Signal Processing*, 1st ed. Cambridge: Cambridge University Press, 2011.

[20] "WTF is a Tensor?!?" Available online at: https://www.kdnuggets.com/2018/05/wtf-tensor.html (Accessed: 25 May 2024).

[21] "Java HashMap Load Factor," Available online at: https://www.baeldung.com/java-hashmap-load-factor (Accessed: 25 May 2024).

[22] "OpenMP Application programming Interface - Version 4.5," Available online at: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (Accessed: 25 May 2024).

[23] E. S. R. van der Pas and C. Terboven, *Using OpenMP – The Next Step. Affinity, Accelerators, Tasking, and SIMD.* The MIT Press, 2017.

[24] "Convolution," Available online at: https://en.wikipedia.org/wiki/Convolution (Accessed: 25 May 2024).