Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño e Implementación del Algoritmo GPU-DBSCAN para la Identificación y Clasificación de Agregados en Imágenes.

Autor: Rodrigo Lomba Moreno
Tutor(a): Nazario Félix González
Cotutor: Antonio Díaz Pozuelo

Madrid, enero 2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Diseño e Implementación del Algoritmo GPU-DBSCAN para la Identificación y Clasificación de Agregados en Imágenes.
Enero 2026

Autor: Rodrigo Lomba Moreno

Tutor:
Nazario Félix González
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
ETSI Informáticos
Universidad Politécnica de Madrid

Cotutor:
Antonio Diaz Pozuelo
Instituto de Química Física Blas Cabrera
Consejo Superior de Investigaciones Científicas

Resumen

Este trabajo presenta el diseño e implementación de una versión paralela del algoritmo DBSCAN dentro del paradigma SIMT, orientada a la identificación y clasificación de agregados en conjuntos de datos. El objetivo principal es acelerar las fases más costosas del algoritmo aprovechando el modelo de ejecución de GPU. En una primera fase se desarrolla una versión secuencial de referencia en Python, seguida de una versión optimizada (vectorizada) en CPU mediante Numba, y finalmente una versión paralela en GPU complementada con una optimización adicional en C para mejorar el rendimiento en la fase de propagación del agregado.

El código incluye además un módulo de extracción de características morfológicas que describe cada agregado identificado, proporcionando diversas métricas, tales como centros geométricos, tamaños y factores de forma elementales. La validación se realiza utilizando tanto datos e imágenes de agregados generados mediante simulaciones de dinámica molecular, así como tratando imágenes fotográficas de muestras biológicas. En ese proceso se evalúa la precisión en la detección de agregados y la coherencia de las propiedades extraídas. Los resultados muestran mejoras significativas en el tiempo de ejecución respecto al algoritmo secuencial de referencia, y a la versión mejorada con Numba.

Asimismo, se analiza el consumo energético de cada implementación, mostrando que la reducción del tiempo en GPU se traduce en un menor consumo compensando el mayor gasto energético de GPU frente a CPU. El trabajo concluye con un análisis de las limitaciones detectadas, así como con propuestas de mejora que incluyen la incorporación de técnicas de preprocesado de imágenes, el uso de modelos basados en redes neuronales para una identificación más robusta y la integración de estructuras espaciales como *linked cells* para mejorar la escalabilidad.

Abstract

This paper presents the design and implementation of a parallel version of the DBSCAN algorithm within the SIMT paradigm, aimed at identifying and classifying clusters in datasets. The main objective is to accelerate the most costly phases of the algorithm by leveraging the GPU execution model. In the first phase, a sequential reference version is developed in Python, followed by an optimised (vectorised) version on CPU using Numba, and finally a parallel version on GPU complemented with additional optimisation in C to improve performance in the cluster propagation phase.

The code also includes a morphological feature extraction module that describes each identified aggregate, providing various metrics such as geometric centres, sizes, and elementary shape factors. Validation is performed using both data and images of aggregates generated by molecular dynamics simulations and by processing photographic images of biological samples. This process evaluates the accuracy of aggregate detection and the consistency of the extracted properties. The results show significant improvements in execution time compared to the sequential reference algorithm and the version enhanced with Numba.

The energy consumption of each implementation is also analysed, showing that the reduction in GPU time translates into lower consumption, offsetting the higher energy consumption of the GPU compared to the CPU. The work concludes with an analysis of the limitations detected, as well as proposals for improvement that include the incorporation of image pre-processing techniques, the use of neural network-based models for more robust identification, and the integration of spatial structures such as linked cells to improve scalability.

Agradecimientos

En primer lugar, deseo expresar mi más sincero agradecimiento a mi cotutor, Antonio Díaz Pozuelo, por su paciencia durante todas las jornadas en las que trabajé en su despacho, por responder siempre a mis preguntas y por acompañarme y orientarme en cada etapa del proyecto. Su apoyo constante ha contribuido de manera decisiva a que este trabajo haya resultado más ameno y accesible.

Asimismo, quiero agradecer a mi familia y amigos su presencia incondicional, su ánimo y su ayuda a lo largo de todo este proceso.

Tabla de contenidos

1 Introducción

En este capítulo se presentarán el contexto y la motivación que ha llevado en la realización de este Trabajo de Fin de Grado. A su vez se describirán los objetivos específicos enumerando las tareas necesarias para su consecución. Por último, se detallará la organización que seguirá este documento.

1.1 Motivación y Contexto

El análisis de imágenes es una disciplina imprescindible en campos como la biología celular, la astronomía o la visión artificial. En éstos y muchos otros campos, la identificación y caracterización de estructuras resulta esencial. Sin embargo, el procesamiento de grandes volúmenes de imágenes con alta resolución supone un desafío computacional considerable, especialmente a la hora de emplear técnicas de agrupamiento o clustering en las que una implementación simplista puede tener una complejidad computacional del tipo $O(N^2)$ o incluso $O(N^2 \log N)$, siendo N el número de pixels.

Entre los algoritmos de clustering, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) destaca por su capacidad para identificar agrupaciones sin la necesidad de conocer su número previamente. No obstante, su aplicación con un conjunto de datos masivos se ve muy limitada por la complejidad computacional, $O(N^2)$, que en implementaciones secuenciales se vuelve inviable.

La irrupción de las GPUs (Graphics Processing Units) ha revolucionado el enfoque de problemas complejos mediante métodos computacionales intensivos. Arquitecturas modernas de GPU, caracterizadas por miles de núcleos, memoria GDDR (Graphics Double Data Rate) con capacidades de transferencia muy superiores a la memoria convencional de las CPUs y unidades especializadas en cálculo numérico, nos ofrecen la posibilidad de acelerar algoritmos tradicionalmente secuenciales. Investigaciones recientes [1], [2] han demostrado la posibilidad de adaptar (paralelizar) el algoritmo DBSCAN para GPUs, logrando aceleraciones superiores a 100x respecto a implementaciones secuenciales.

De ahí la necesidad de explorar esta vía para optimizar mediante los recursos de GPU ahora ampliamente disponibles, en qué medida es posible implementar algoritmos que aprovechen el enorme potencial de paralelización de las GPUs actuales.

1.2 Objetivos del Trabajo

En línea con lo mencionado en el apartado anterior, este Trabajo de Fin de Grado tiene como objetivo diseñar e implementar una versión del DBSCAN paralela en GPU orientada específicamente a la identificación y clasificación de agregados en imágenes. La estrategia de paralelización se centra en optimizar los cálculos aprovechando el modelo de ejecución SIMT (Single Instruction Multiple Thread) y las jerarquías de memoria implementadas en la arquitectura CUDA. Además, se integrará un módulo de extracción de características para describir mínimamente la morfología de cada agrupación identificada.

En este trabajo, se comparará el rendimiento del algoritmo de GPU frente al correspondiente secuencial en CPU, evaluando la eficiencia, escalabilidad,

precisión en la identificación de agregados y la efectividad del módulo de extracción de características.

Para alcanzar esta meta, se han definido los siguientes objetivos específicos:

- **Formalización del problema. Elementos básicos:**
  - **Comprensión del algoritmo DBSCAN y sus aplicaciones.** Mediante el estudio del fundamento teórico, sus parámetros y su comportamiento en diferentes contextos.
  - **Dominio de técnicas básicas de programación paralela basada en CUDA.** Analizando la arquitectura de las GPUs, el modelo de programación de hilos, la gestión de los diferentes tipos de memoria y la sincronización.
- **Desarrollo de una primera versión secuencial del algoritmo DBSCAN.** Implementando una versión en Python, que servirá para la comprobación y comparación de resultados.
- **Implementación de una versión paralela en GPU de DBSCAN.** Diseñando y desarrollando kernels CUDA que paralelicen las fases necesarias, buscando reducir el costo computacional del algoritmo.
- **Desarrollo de un módulo para la obtención de un conjunto de características mínimas de los agregados.** Programación de funciones para extraer las métricas como el centro y la forma de cada agregado, tanto en la versión secuencial como en la paralela.
- **Optimización de las versiones secuencial y paralela.** Mejorar ambas implementaciones analizando el rendimiento y aplicando optimizaciones algorítmicas y estructurales con el objetivo de reducir el tiempo de ejecución computacional.
- **Evaluación comparativa (benchmarking) de ambas implementaciones.** Realización del diseño de un conjunto de experimentos, seleccionando los datos de entrada y las métricas objetivas para evaluar las ventajas de la paralelización.
- **Puesta a prueba del código aplicándolo al reconocimiento y clasificación de agregados obtenidos a partir de imágenes experimentales.** Mediante el uso de datos experimentales para demostrar la utilidad práctica del algoritmo en un escenario público.

Con la implementación de estos objetivos específicos se conseguirá un sistema GPU-DBSCAN funcional y optimizado, capaz de acelerar el proceso de agregación de imágenes.

1.3 Tareas y Planificación del Trabajo

Durante el desarrollo de este Trabajo de Fin de Grado se ha utilizado un enfoque metodológico basado en tareas, que ha sido estructurado en fases secuenciales.

A continuación, se enumeran las principales fases, así como las tareas relacionadas:

Fase 1: Investigación y análisis (semanas 1-4):

- Revisión bibliográfica inicial: Consulta de artículos y documentación para entender el funcionamiento del algoritmo DBSCAN.

- Estudio de la computación paralela en GPU: Análisis de la arquitectura CUDA y de las herramientas necesarias para programar sobre GPU.

Fase 2: Diseño e Implementación de la versión secuencial del algoritmo (semanas 3-7):

- Desarrollo de una versión secuencial del algoritmo: Implementación de DBSCAN en CPU para luego compararlo con la versión en GPU.
- Optimización y pruebas iniciales en CPU: Ajuste del código y comprobación de su correcto funcionamiento en pruebas sencillas.

Fase 3: Diseño e Implementación de la versión paralela en GPU del algoritmo (semanas 6-11):

- Implementación de la versión paralela GPU-DBSCAN: Adaptación del algoritmo para que ejecute las partes más costosas en una GPU, explorando la paralelización de cálculos y optimizando el uso de memoria.
- Optimización y pruebas iniciales en GPU: Ajuste del código y comprobación de su correcto funcionamiento en pruebas sencillas y con los resultados obtenidos en las mismas pruebas en CPU.
- Diseño de pruebas en GPU: Desarrollo de pruebas y elección del conjunto de imágenes que se emplearan.

Fase 4: Diseño e implementación del módulo de extracción de características de los agregados (semana 9-11):

- Programación del módulo de extracción de características de los agregados: Desarrollo de un módulo de posprocesamiento que calcula métricas descriptivas para cada agregado identificado por el algoritmo DBSCAN.

Fase 5: Diseño de pruebas Finales (semana 11-13):

- Diseño de pruebas finales en CPU y GPU: Desarrollo de pruebas y elección del conjunto de imágenes que se emplearan.

Fase 6: Documentación (semana 8-16):

- Redacción de la documentación técnica y del informe final del TFG: Elaboración de un documento detallando el diseño del algoritmo, la metodología y las pruebas realizadas. Así como un informe de todo el trabajo realizado, resultados y conclusiones.
- Revisión y correcciones finales: Ajuste y edición de la documentación y del informe antes de la entrega.

La figura 1 muestra el diagrama de Gantt que refleja la planificación de las tareas descrita anteriormente.

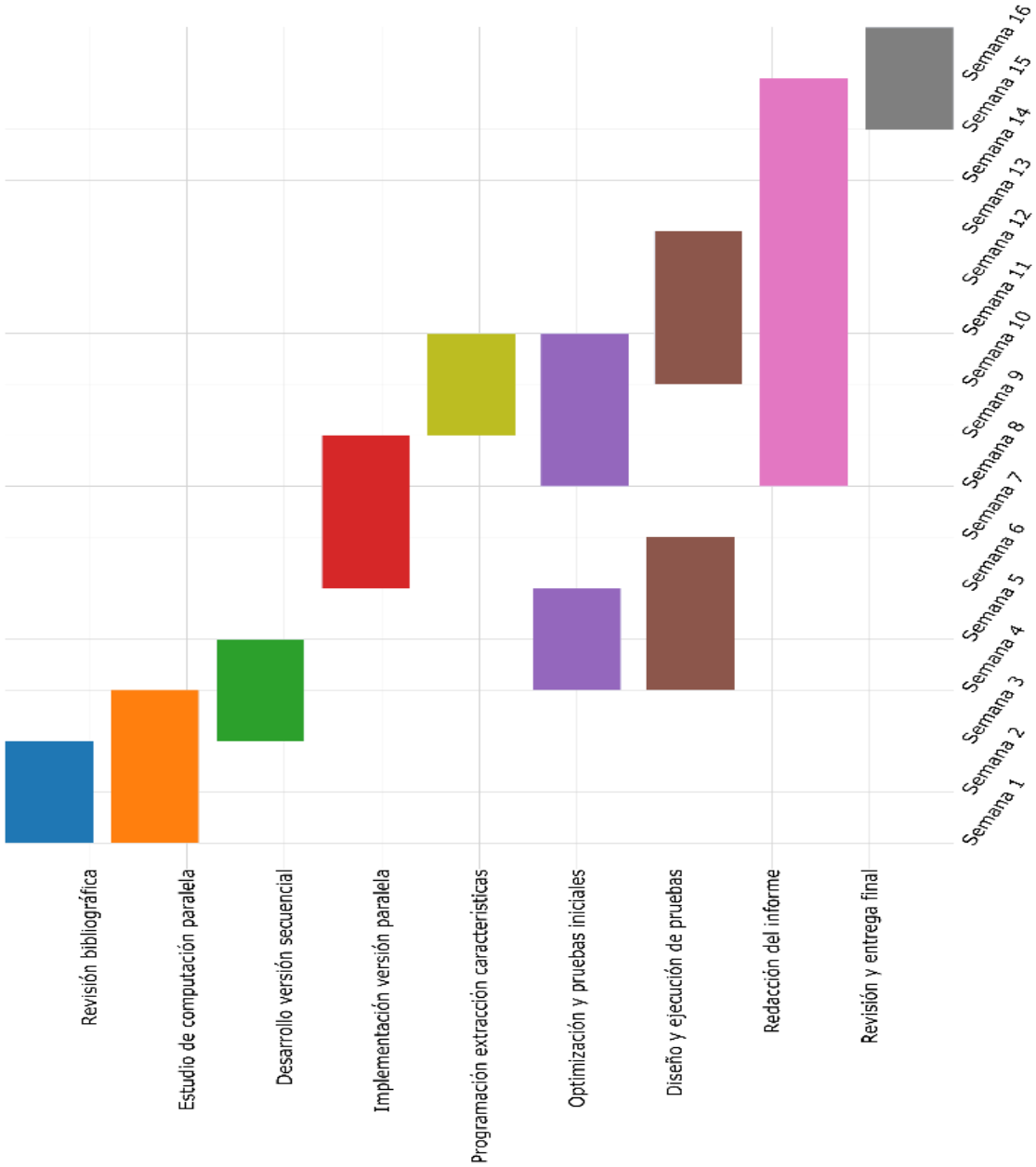


Figura 1: Diagrama de Gantt.

1.4 Estructura del Documento

A continuación, se presenta una breve descripción del contenido del documento, que consta de ocho capítulos, bibliografía y anexos:

- **Capítulo 1 - Introducción:** se presenta la motivación y el contexto en el que se enmarca el proyecto, los objetivos del trabajo, la planificación y la estructura del documento.
- **Capítulo 2 - Estado del Arte:** se revisan los fundamentos teóricos del algoritmo DBSCAN y sus aplicaciones, los conceptos necesarios de computación en GPU y arquitectura CUDA, así como las técnicas de extracción de características morfológicas en agregados utilizadas.
- **Capítulo 3 - Diseño y Arquitectura del Sistema:** se explica cómo está diseñado el código. Esto incluye cómo se procesan los datos que se introducen, cómo funciona el algoritmo, el cálculo de propiedades y se identifican las partes importantes y los posibles problemas que se pueden encontrar.
- **Capítulo 4 - Implementación del Sistema:** se explica qué decisiones se tomaron para implementar el sistema, qué entorno de desarrollo se usó y qué optimizaciones se hicieron.
- **Capítulo 5 - Validación de la Implementación:** se presenta la metodología de validación, describiendo los conjuntos de datos utilizados y los criterios de evaluación.
- **Capítulo 6 - Pruebas, Resultados y Análisis:** se presentan los resultados que se obtuvieron al realizar las pruebas. También se examinan los tiempos que tomaron estas pruebas para ejecutarse. Además, se analiza la mejora en el tiempo de ejecución que se logra cuando se comparan las versiones paralelas con las versiones que se ejecutan en la CPU.
- **Capítulo 7 - Conclusiones:** se recogen las conclusiones generales del trabajo.
- **Capítulo 8 – Análisis de Impacto:** se examina el impacto potencial del proyecto y se relaciona el trabajo con los Objetivos de Desarrollo Sostenible.
- **Bibliografía:** incluye las referencias usadas.
- **Anexos:** contiene información adicional y materiales complementarios, como el repositorio del proyecto y el informe de originalidad.

Mediante esta organización del documento se garantiza una presentación coherente y completa del proyecto.

2 Estado del Arte

En este capítulo se presenta una revisión de los fundamentos teóricos que sustentan el desarrollo de este trabajo. En él se analizan los principios del algoritmo DBSCAN y sus aplicaciones, en segundo lugar, se explora la arquitectura de las GPUs y el modelo de programación CUDA. Finalmente, se

revisarán las técnicas de extracción morfológicas aplicadas a resultados de agregación. Una vez presentados estos conceptos, se dispondrá de las nociones básicas necesarias para el desarrollo del código.

2.1 Algoritmo DBSCAN: Fundamentos y Aplicaciones

El algoritmo DBSCAN (Density-Based Spatial Clustering of Applications with Noise) es una técnica de agregación basada en la densidad, que permite identificar grupos en conjuntos de datos [3]. Su principal ventaja reside en su capacidad para detectar agregados sin requerir la especificación previa de su número. La idea fundamental detrás de este enfoque es que los agregados se encuentran en regiones con alta densidad de puntos, separadas por áreas de baja densidad.

2.1.1 Conceptos

El algoritmo utiliza dos parámetros principales: ε, que define el radio de búsqueda para determinar la vecindad de cada punto, y MinPts, que establece el número mínimo de puntos requeridos para considerar una agrupación de puntos como tal. Este radio, es una distancia, que bien puede ser euclídea en un espacio n-dimensional (tal es el caso que nos ocupa en el tratamiento de imágenes), o cualquier otra métrica que nos permita agrupar conjuntos de datos en términos de una densidad. El algoritmo se sustenta en varios conceptos clave, comenzado por la noción de puntos core o núcleo: éste se define como un punto que tiene un número mayor o igual de puntos, MinPts-1, situados a una distancia menor o igual a ε: un círculo (o esfera n-dimensional) de radio ε centrada en un núcleo contiene MinPts puntos. Los puntos bordes son aquellos que se encuentran dentro del ε de un punto core pero no cumplen la condición de MinPts. Finalmente, los puntos ruido son aquellos que no pertenecen a ninguna de las categorías anteriores, y por tanto no forman parte de ningún agregado. En la figura 2 se muestra un ejemplo de estos conceptos.

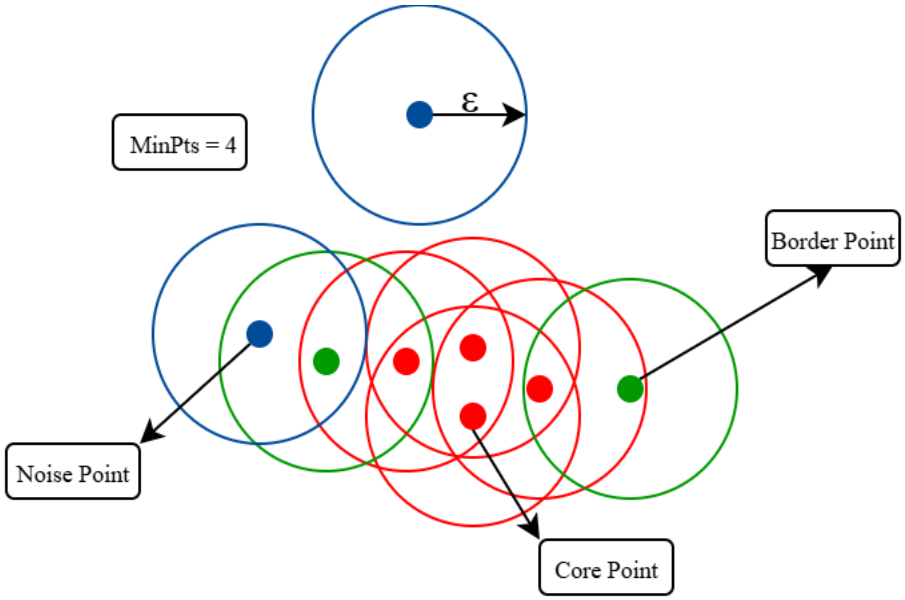


Figura 2. Ejemplo DBSCAN con MinPts = 4.

La condición fundamental para que dos puntos pertenezcan al mismo agregado es que estén conectados a través de una secuencia de puntos núcleo. Un punto borde puede pertenecer a un agregado si es vecino de al menos un punto núcleo del agregado, pero la conexión entre puntos bordes siempre debe estar mediada por puntos núcleo. Esta definición permite que los agregados con formas arbitrarias sean identificados correctamente.

2.1.2 Funcionamiento

En la práctica, el algoritmo DBSCAN se reduce a los siguientes pasos:

1. Inicialización: todos los puntos del conjunto de datos se marcan como no visitados.

2. Selección del punto inicial: se elige un punto no visitado p y se marca como visitado.

3. Búsqueda de vecinos: se obtienen los puntos situados a una distancia menor o igual que ε de p.
   - Si el número de vecinos es mayor o igual que MinPts (incluyendo al propio p), el punto se considera punto núcleo y se inicializa un nuevo agregado C añadiendo a p.
   - Si no cumple esta condición, p se marca temporalmente como ruido y se selecciona otro punto no visitado.

4. Propagación inicial: todos los vecinos de p se añaden a una lista de puntos pendientes de análisis.

5. Propagación del agregado: para cada punto q de la lista:
   - Si q no ha sido visitado, se marca como visitado y se calculan sus vecinos.
   - Si q es un punto núcleo, todos sus vecinos se incorporan a la lista de análisis, ampliando así el agregado.
   - Si q no pertenece todavía a ningún agregado, se añade a C.

6. Finalización del agregado: cuando la lista de puntos pendientes queda vacía, el agregado C se considera completo.

7. Continuación del proceso: el algoritmo continúa seleccionando nuevos puntos no visitados y repitiendo el procedimiento hasta que todos los puntos hayan sido procesados.

8. Reevaluación del ruido: algunos puntos marcados inicialmente como ruido pueden convertirse en puntos frontera si resultan estar dentro de ε de algún punto núcleo, por lo que se revisan durante el proceso.

9. Asignación final: los puntos que no se incorporan a ningún agregado se etiquetan definitivamente como ruido.

2.1.3 Aplicaciones

La efectividad de DBSCAN se ha confirmado en diversos campos. En el ámbito de la detección de anomalías, se ha aplicado con éxito en redes de sensores inalámbricos [4], en el campo de las ciencias ambientales (en tareas de limpieza

de datos hidrológicos y de calidad del agua [5]), así como en otros campos: como el sector financiero (para la identificación de transacciones financieras sospechosas [6]) y la bioinformática [7]; lo que demuestra su flexibilidad y utilidad.

2.2 Computación en GPU y Arquitectura CUDA

En esta sección, se introduce el concepto fundamental de paralelismo, seguidos de una descripción de la arquitectura CUDA. Estos conceptos son necesarios para implementar algoritmos complejos de manera eficiente aprovechando todos los beneficios ofrecidos por las GPUs.

2.2.1 Paralelismo

El paralelismo [8] en computación consiste en la ejecución simultánea de múltiples tareas, permitiendo realizar operaciones de forma independiente, tal y como se muestra en el siguiente diagrama:

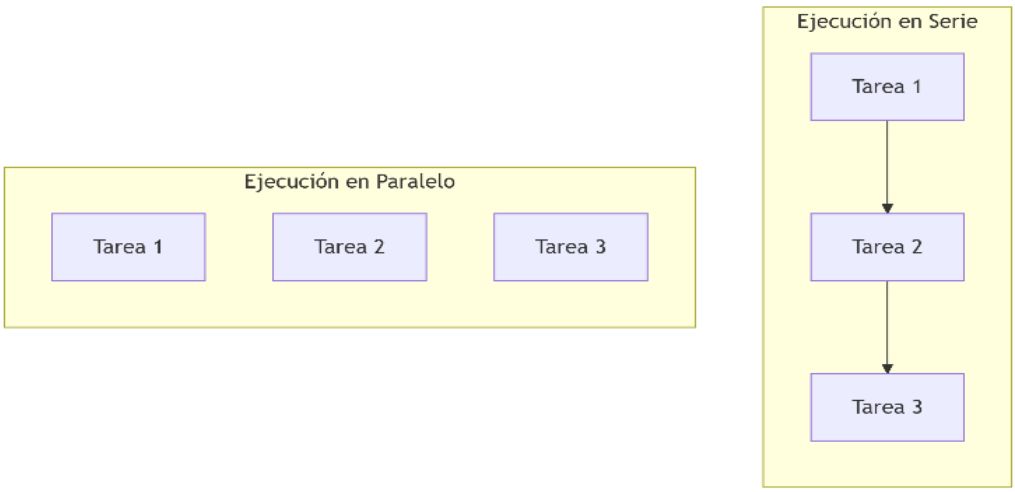


Figura 3: Diferencias entre Ejecución en Serie y en Paralelo

Su implementación en CPU ha sido ampliamente desarrollada desde el siglo pasado, tanto en su variante de memoria compartida (Posix Threads (*pthreads*), OpenMP) como distribuida (MPI en sus múltiples implementaciones). El hardware peculiar de las GPU ha dado lugar a un nuevo paradigma de programación que se explica brevemente a continuación.

2.2.2 Beneficios de las GPU

En el ámbito de la computación de alto rendimiento, la elección entre CPU y GPU viene determinada por la carga de trabajo y la eficiencia. Para aplicaciones con un alto grado de paralelismo, las GPUs ofrecen ventajas en el rendimiento computacional y el ancho de banda de memoria. En los últimos 20 años las GPUs se han convertido en procesadores con altas capacidades de paralelización, con miles de núcleos para trabajo concurrente, a los que recientemente se han incorporado los Tensorcores, con capacidad de procesado matricial. [9]

La principal diferencia de capacidad computacional entre las CPUs y las GPUs se encuentra en el diseño. Mientras la CPU está diseñada para ejecutar una secuencia de operaciones lo más rápido posible, la GPU está diseñada para tareas de cómputo intensivo y altamente paralelizable. Es por ello por lo que se las diseña de forma que los transistores están más dedicados al procesamiento de datos en vez de al caché de datos y al control de flujo.

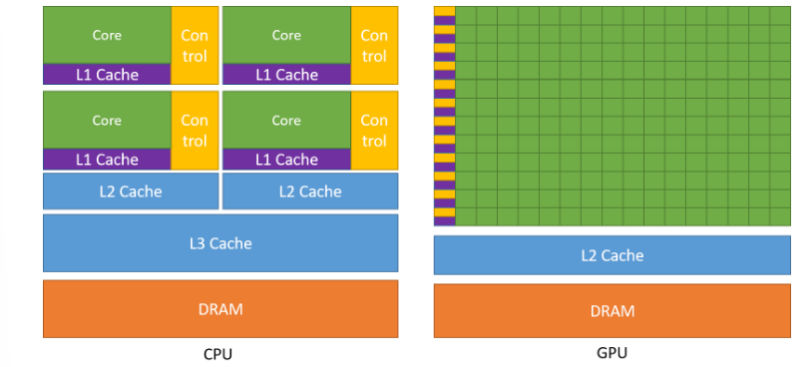


Figura 4: Diferencias entre los transistores de CPU y GPU. [9]

Como se muestra en la Figura 4, esta distribución de recursos permite a la GPU un rendimiento significativamente mayor en operaciones paralelas. La arquitectura GPU, organizada alrededor de múltiples núcleos altamente eficientes, puede ejecutar miles de hilos concurrentemente, logrando una mejora sustancial en el rendimiento computacional frente a las CPUs para cargas de trabajo altamente paralelizables. Esta ventaja se manifiesta especialmente en aplicaciones que requieren un procesamiento masivo de datos, donde el alto ancho de banda de memoria de las GPUs se combina con su capacidad de procesamiento paralelo alcanzando un rendimiento muy superior a los posibles con las CPU tradicionales. Un factor que contribuye esencialmente a potenciar esta capacidad, es el mínimo coste computacional de creación de un hilo y cambio de contexto en una GPU en comparación con una CPU, por el propio diseño del hardware.

2.2.3 CUDA

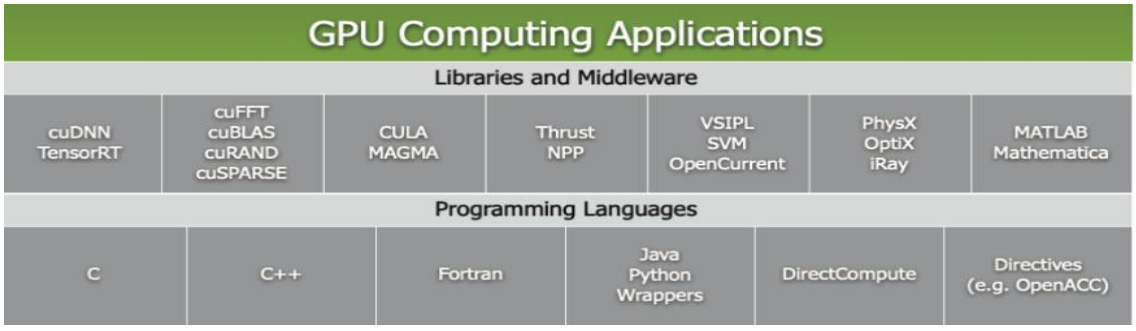


Figura 5: Principales bibliotecas y lenguajes de programación compatibles con CUDA. [9]

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y modelo de programación desarrollado por NVIDIA en 2006, que permite usar las GPU para computación de propósito general. Gracias a esta arquitectura, los desarrolladores pueden acceder directamente a los recursos de la GPU mediante extensiones de lenguajes de programación estándar. Aunque el SDK (Software Development Kit) está pensado para programar en C++ como lenguaje de programación de alto nivel, actualmente se soportan otros leguajes e interfaces de programación, como se muestra en la Figura 5.

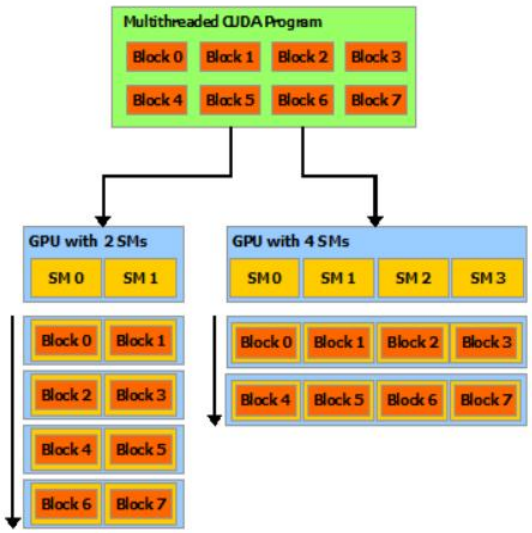


Figura 6: Distribución automática de bloques de hilos en GPUs con diferente número de Multiprocesadores de Streaming (SMs). [9]

Una de las principales características del modelo CUDA es su escalabilidad automática. Gracias a esta característica, sin que el programador modifique el código las aplicaciones aprovechan de forma natural y eficiente el hardware, ya que el sistema de ejecución se encarga automáticamente de distribuir los bloques entre los SMs (multiprocesadores). De esta manera, un mismo programa puede ejecutarse en cualquier GPU compatible, independientemente del número de SMs que tenga, tal como se muestra en la Figura 6.

2.2.4 Kernels

En el modelo de programación CUDA, el concepto *Kernel* es fundamental. Un *Kernel* es una función que se define en el código del host (CPU) pero se ejecuta de forma paralela en el dispositivo, (*device* GPU). Cuando se lanza un kernel, se genera una gran cantidad de hilos (*threads*) que ejecutan simultáneamente el mismo código, pero cada uno sobre una porción diferente de los datos, siguiendo el paradigma SIMT (*Single Instruction, Multiple Thread*)

La invocación de un *kernel* se realiza mediante una sintaxis especial donde se especifica la configuración de la ejecución. Esta configuración define una estructura jerárquica (Figura 7) compuesta por:

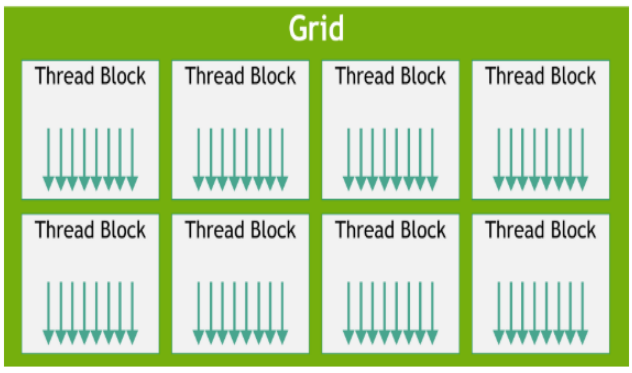


Figura 7: Estructura de una malla (Grid) en CUDA, compuesta por múltiples bloques de hilos (Thread Blocks). [9]

- **Hilos (*Threads*):** Unidad básica de ejecución. Cada hilo ejecuta una instancia del *kernel* y tiene un identificador único.
- **Bloques (*Blocks*):** Conjunto de hilos que se ejecutan en el mismo multiprocesador (SM) y pueden cooperar entre sí mediante memoria compartida y sincronización.
- **Mallas (*Grids*)**: Conjunto total de bloques que se lanzan para la ejecución de un *kernel*.

En las arquitecturas más recientes, se ha introducido un nivel adicional, el *Thread Block Clusters*, que son agrupaciones de múltiples bloques de hilos (*clusters*) que permiten una cooperación y sincronización a mayor escala, permitiendo el uso de memoria compartida entre los bloques que forman parte de un *cluster*, limitado por el momento a un máximo de 8 bloques por *cluster*.

2.2.5 Memoria

La arquitectura CUDA implementa un sistema de memoria jerárquico para maximizar el ancho de banda y minimizar la latencia en el acceso a datos. Cada nivel ofrece diferentes compromisos entre la velocidad, capacidad y alcance, lo que permite optimizar el rendimiento en función del patrón de acceso.

Como se muestra en la Figura 8, la jerarquía de memoria se organiza en:

- **Memoria Local**: memoria privada para cada hilo individual. Se almacena en registros o en la memoria global. Óptima para variables temporales y datos de uso frecuente dentro de un hilo.
- **Memoria Compartida**: memoria de baja latencia ubicada en cada multiprocesador. Es compartida por todos los hilos de un mismo bloque, permitiendo la comunicación y cooperación entre ellos. Su capacidad es limitada pero su velocidad la hace ideal para la reutilización de datos.
- **Memoria Global**: Es la memoria principal de la GPU, de mayor capacidad, pero también mayor latencia. Es accesible por todos los hilos de todos los bloques y persiste entre diferentes lanzamientos de *kernels*.
- **Thread Block Cluster**: Como se ha explicado recientemente, en las arquitecturas más nuevas se han introducido los *clusters* de bloques que permiten compartir memoria entre los bloques de cada *cluster*.

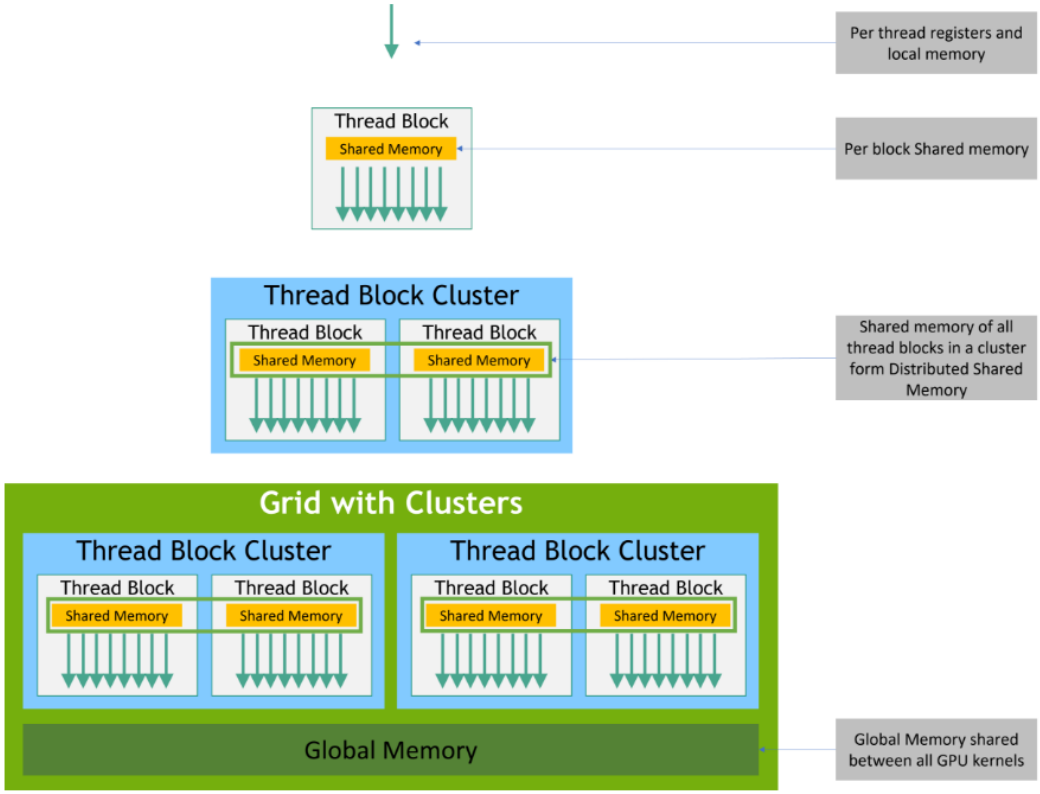


Figura 8: Jerarquía de memoria en CUDA mostrando los diferentes niveles. [9]

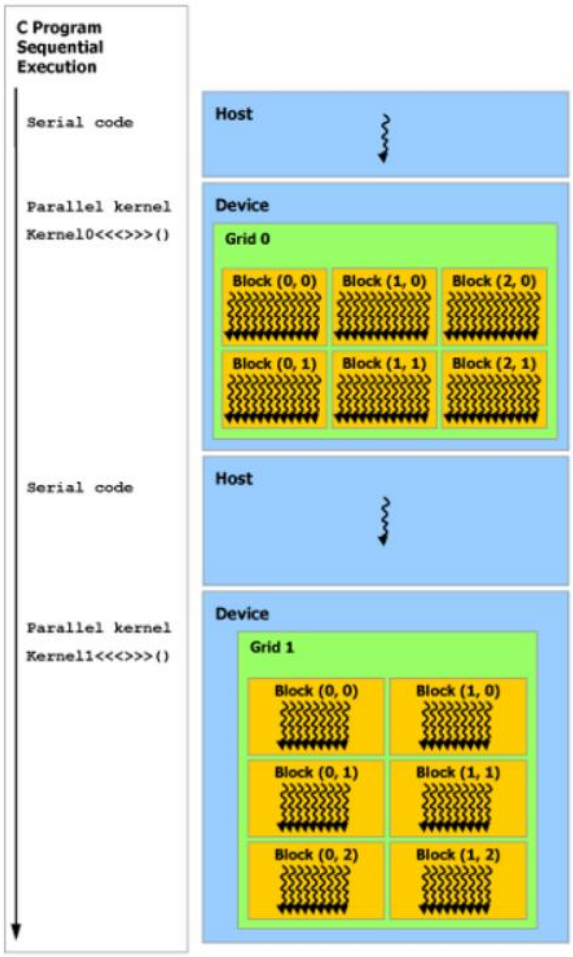
La gestión eficiente de esta jerarquía es crucial para el rendimiento. Estrategias como el uso de la memoria compartida, la correcta alineación de datos

(coalescencia) y la organización de los accesos a la memoria global, pueden resultar en mejoras de rendimiento muy significativas. Estas optimizaciones permiten que los hilos contiguos accedan a posiciones de memoria igualmente contiguas. Por lo tanto, la GPU podrá agrupar las solicitudes de lectura en transacciones de mayor tamaño (típicamente de 128 bits) en lugar de realizar múltiples lecturas independientes; lo que conlleva menores operaciones de lectura maximizando el ancho de banda de acceso a memoria.

2.2.6 Programación heterogénea

El modelo de programación CUDA se fundamenta en un concepto de computación donde el sistema utiliza de manera coordinada tanto la CPU como la GPU para ejecutar una aplicación. Con este enfoque se optimiza al máximo la ejecución de un programa aprovechando las mejores características de ambos, usando la CPU para tareas de control, toma de decisiones y ejecución secuencial, y la GPU para procesamiento paralelo masivo.

Esta colaboración suele seguir un flujo de ejecución bien definido (Figura 9):



1. La CPU prepara los datos de entrada, configura el entorno de ejecución y realiza la transferencia de datos desde su memoria al espacio de memoria de la GPU.
2. A continuación, se llama a los *kernel*, que se ejecutarán en la GPU de manera asíncrona, devolviendo el control del programa a la CPU una vez lanzados.
3. Finalmente, la CPU realiza la transferencia de los datos desde la GPU a la CPU. Habiendo esperado a la finalización del *kernel*. Una vez se han procesado los datos de la GPU el programa en serie genera la salida.

Figura 9: Flujo de ejecución típico de un programa CUDA. [9]

Una vez explicados las fases por las que pasa es importante recalcar que para maximizar el rendimiento en este modelo se debe minimizar las transferencias de datos entre GPU y CPU debido a su alto coste y gestionar adecuadamente la

jerarquía de memorias, por ello el programador debe transferir los datos a la GPU, mantenerlos en ella y centrarse en reutilizarlos. A su vez el programador siempre debe proporcionar suficiente trabajo a la GPU, ya que poco trabajo significa desperdicio en ciclos ociosos (*idle cycles*). Finalmente, unos de los aspectos más importantes que tiene que seguir el desarrollador es la coalescencia de datos, es decir hay que procurar que el acceso de datos, en *arrays*, sea (siempre que se pueda) en posiciones contiguas de la memoria. Un acceso inadecuado a la memoria en *arrays* de gran tamaño puede degradar el rendimiento varios órdenes de magnitud, un ejemplo básico es la necesidad de transponer una matriz antes de llevar a cabo un producto de matrices elemento a elemento.

2.3 Extracción de Características Morfológicas en Agregación

Una de las etapas fundamentales en el análisis de agregados es la caracterización morfológica, que permite cuantificar propiedades geométricas. [10]

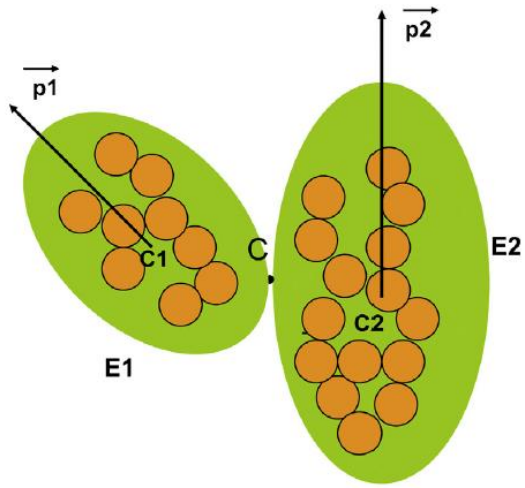


Figura 10: Ejes principales de inercia de agregado. [10]

El primer paso en el análisis morfológico consiste en determinar la posición del **centro de masas** (CDM), o centro geométrico, de cada agregado. Para un sistema discreto compuesto por N_{clus} partículas, el CDM se calcula como:

$$\overrightarrow{R_{\text{CDM}}} = \frac{1}{N_{\text{clus}}} \sum_{i=1}^{N_{\text{clus}}} \vec{r_i}$$

donde $\vec{r_i}$ son las coordenadas de cada partícula dentro del agregado.

A partir del CDM, calculamos el **radio de giro,** R_g , una magnitud ampliamente utilizada en la caracterización de polímeros y agregados con formas compactas, y que nos proporciona una medida del tamaño efectivo del agregado [11]:

$$R_g^2 = \frac{1}{N_{\text{clus}}} \sum_{i=1}^{N_{\text{clus}}} \left| \vec{r_i} - \overrightarrow{R_{\text{CDM}}} \right|^2$$

Por otro lado, para caracterizar la forma de los agregados se emplea el **tensor de inercia** (para nuestros propósitos las *masas* son idénticas e iguales a la unidad), definido en un sistema de coordenadas con origen en el CDM. Para un agregado en 2D, el tensor viene dado por:

$$\mathcal{I} = \begin{pmatrix} \sum y_i'^2 & -\sum x_i' y_i' \\ -\sum x_i' y_i' & \sum x_i'^2 \end{pmatrix}$$

donde $\vec{r}'_i = \vec{r}_i - \vec{R}_{CDM}$.

Una vez obtenemos el tensor de inercia, su diagonalización es crucial para obtener los **valores propios λ_1 y λ_2** :

$$\lambda_{1,2} = \frac{\mathcal{I}_{xx} + \mathcal{I}_{yy} \pm \sqrt{\left(\mathcal{I}_{xx} - \mathcal{I}_{yy}\right)^2 + 4\mathcal{I}_{xy}^2}}{2}$$

Dejando el tensor diagonalizado como:

$$\mathcal{I}_{\text{diag}} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

Con estos valores la interpretación que podemos tener es:

- Si $\lambda_1 \approx \lambda_2$, la forma es aproximadamente circular.
- Si $\lambda_1 \gg \lambda_2$ o $\lambda_1 \ll \lambda_2$, el agregado tendrá una forma alargada.

Una vez definida la forma del agregado nos interesa saber la desviación que hay de la forma obtenida respecto a la forma ideal, es por ello que necesitaremos:

- **Desviación de forma esférica** (para los agregados con forma circular):

$$\Delta_{\text{sph}} = \sqrt{\frac{\left(\lambda_1 - \text{Tr}[\mathcal{I}_{\text{diag}}]\right)^2 + \left(\lambda_2 - \text{Tr}[\mathcal{I}_{\text{diag}}]\right)^2}{\lambda_1^2 + \lambda_2^2}}$$

  donde *Tr[X]* es la traza de una matriz, suma de los elementos de su diagonal principal.

- **Desviación de forma cilíndrica** (para los agregados con una forma alargada):

$$\Delta_{\text{cyl}} = \frac{\lambda_2 - \lambda_1}{\sqrt{\lambda_1^2 + \lambda_2^2}}$$

  Estos descriptores son de gran utilidad para clasificar y seleccionar agregados en función de su morfología y las necesidades específicas del estudio.

3 Diseño y Arquitectura del Código

El presente capítulo describe el diseño y la arquitectura del código necesarios para adaptar el algoritmo DBSCAN a una ejecución eficiente en GPU. Se analizan sus componentes fundamentales, el procesamiento de los datos de entrada y las decisiones estructurales que permiten transformar el algoritmo original en una versión paralelizable. Además, se detalla el diseño del módulo de extracción de características y se identifican las principales zonas críticas y cuellos de botella que surgen durante la paralelización.

3.1 Procesamientos de Datos de Entrada

El procesamiento de los datos de entrada constituye una fase fundamental dentro del diseño del código, ya que determina cómo se interpretan, transforman y organizan los datos antes de ejecutar el algoritmo. Dado que el sistema admite distintos tipos de ficheros, principalmente imágenes y ficheros NetCDF [12] con coordenadas, habrá que establecer un procedimiento unificado que permita convertir los formatos compatibles de entrada a conjunto de puntos en un espacio bidimensional (x,y). Conviene subrayar que el algoritmo DBSCAN se puede aplicar a cualquier dimensionalidad, pero en nuestra implementación

nos vamos a limitar a imágenes y conjuntos de datos bidimensionales. La generalización a dimensionalidades mayores no es especialmente compleja.

El objetivo de esta fase es garantizar que los datos se encuentren en un estado consistente y estructurado de forma óptima para las operaciones posteriores. Aunque el propósito es común para ambas arquitecturas, la forma en la que se organiza la información presenta diferencias relevantes, especialmente en lo relativo a la disposición en memoria y a las dimensiones de las estructuras derivadas del conjunto de datos, debido a las diferencias arquitectónicas entre CPU y GPU.

Procesamiento de Imágenes.

Cuando la entrada corresponda a una imagen, el código transforma una representación matricial en un conjunto de puntos. Este proceso sigue una secuencia bien definida:

1. **Conversión a escala de grises**, reduciendo la imagen a un único canal de intensidad.
2. **Detección automática del color de fondo**, identificando el valor más frecuente en la imagen.
3. **Cálculo de un umbral adaptativo**, ajustado dinámicamente en función del fondo detectado.
4. **Generación de la imagen binaria**, clasificando cada píxel como fondo u objeto.
5. **Determinación del color de los agregados**, analizando la distribución de valores binarios.
6. **Extracción de coordenadas**, recorriendo los píxeles no pertenecientes al fondo y generando sus posiciones (x, y).

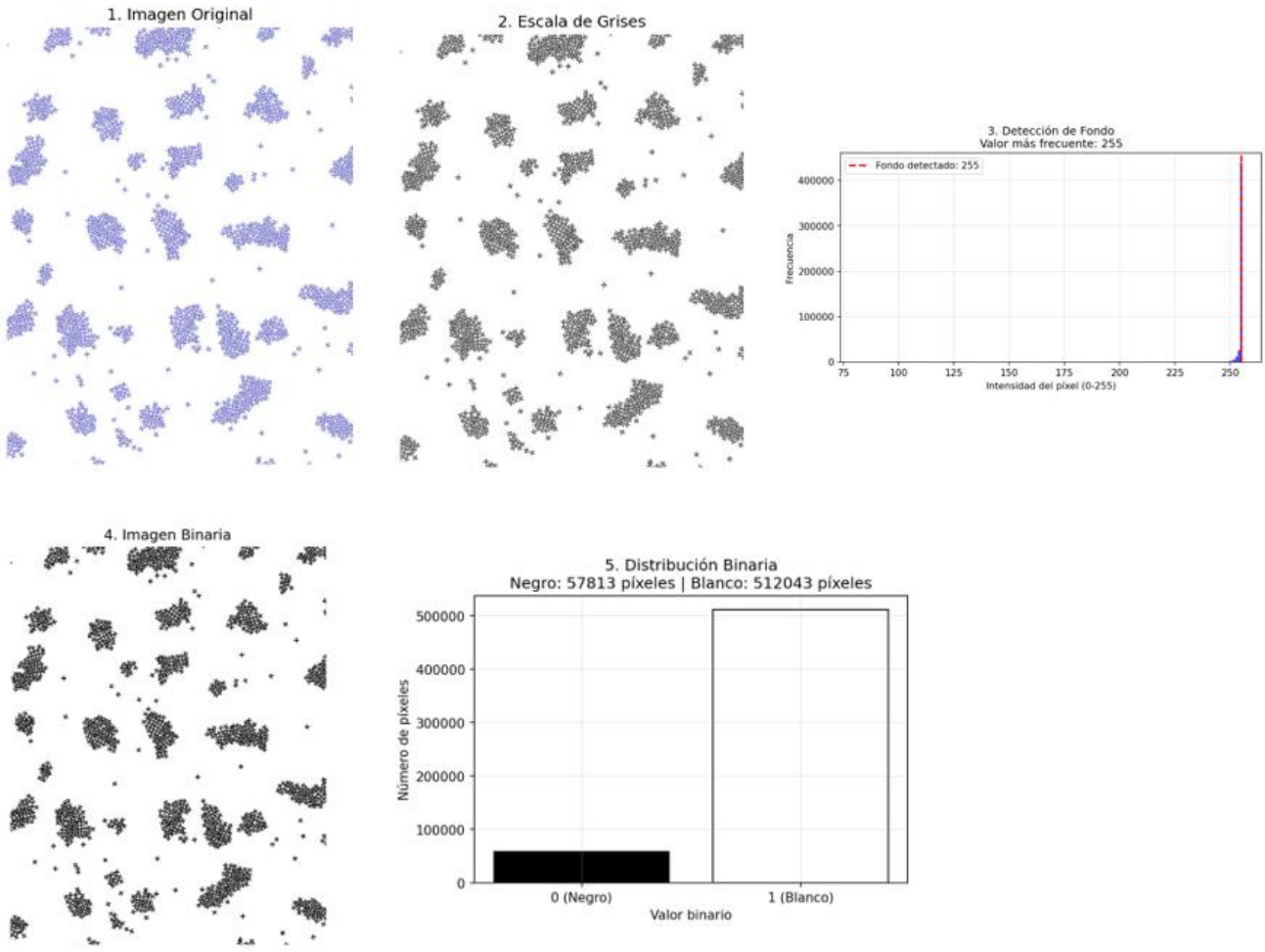


Figura 11: Flujo de Procesamiento de Imagen para Obtención de Puntos.

El resultado será un conjunto de puntos que describe la geometría del objeto presente en la imagen. En CPU este proceso se realiza de forma secuencial, mientras que en GPU se paraleliza completamente, asignando un hilo por píxel para acelerar tanto el recuento como la extracción de coordenadas.

Procesamiento de Ficheros NetCDF

Cuando la entrada corresponda a un fichero NetCDF, el código extrae las coordenadas almacenadas en el *dataset*. Estos ficheros suelen contener información estructurada en forma de matrices multidimensionales, por lo que el primer paso consiste en seleccionar el *frame* adecuado y acceder a las variables que contengan las posiciones espaciales.

Una vez extraídas, las coordenadas se reorganizan para formar un conjunto de puntos bidimensionales contiguos en memoria. En CPU esta reorganización se realiza en memoria dinámica (RAM), mientras que en GPU se lleva a cabo directamente en memoria global (VRAM), evitando transferencias intermedias y permitiendo que los datos estén listos para el procesamiento paralelo desde el primer momento.

Unificación del Formato de Entrada

Independientemente del origen de los datos, el sistema transforma toda la información en un conjunto contiguo de puntos bidimensionales. Esta unificación será imprescindible para garantizar que las fases posteriores del algoritmo puedan ejecutarse sin necesidad de adaptar el formato de los datos.

Aquí aparecen una diferencia importante entre CPU y GPU:

- En CPU, los puntos se almacenan como una estructura bidimensional de tamaño $N \times 2$. El acceso a los componentes mediante una indexación directa (data [i, j]), donde i representa el índice de la estructura y j el componente de la coordenada.
- En GPU, los puntos se almacenan como una estructura lineal de tamaño $2N$, con las coordenadas intercaladas para permitir accesos paralelos y coalescentes. El acceso a los componentes se realiza mediante una indexación lineal o vectorial, i * d + j, donde i representa el índice de la estructura, d la dimensión (en este caso d = 2) y j el componente de la coordenada.

Ambas representaciones contienen la misma información, pero la disposición utilizada en GPU responde a las restricciones propias de su arquitectura, en la que únicamente una disposición de los datos de forma lineal y contigua garantizar accesos paralelos eficientes (coalescencia en memoria).

Resumen

Esta fase finaliza con la obtención de un conjunto de puntos estructurado de manera uniforme, resultado de la transformación de las distintas fuentes de entrada. Con ello se garantiza que el algoritmo DBSCAN dispone de una base de datos clara y consistente sobre la que aplicar sus operaciones posteriores.

3.2 Diseño del Algoritmo DBSCAN

En este apartado se analizará el algoritmo DBSCAN desde la perspectiva del diseño del sistema, centrándose en los elementos que condicionan su futura implementación tanto en CPU como en GPU. Puesto que los fundamentos teóricos del algoritmo ya han sido explicados en el capítulo anterior, aquí se estudiarán únicamente los componentes que influirán en las decisiones arquitectónicas y en la estructura computacional del sistema.

3.2.1 Elección de Parámetros

DBSCAN depende de dos parámetros fundamentales: ε (epsilon), que define el radio de vecindad, y MinPts, que determina el número mínimo de puntos necesarios para que un punto sea considerado *core*. La elección de estos parámetros será determinante para la calidad de la agregación, el tamaño del grafo de vecindad y el coste computacional de las fases posteriores. Por ello, el código permite que ambos parámetros puedan ser proporcionados manualmente por el usuario, pero también incorpora mecanismos automáticos para calcularlos cuando no se especifiquen.

Selección de MinPts.

Dado que los datos procesados serán coordenadas bidimensionales (x, y), se adopta la regla habitual en agregación espacial [3]:

$MinPts = 2 \times dimensión + 1 = 5$

Esta elección proporciona un umbral razonable para identificar regiones densas sin introducir complejidad adicional. Tanto la versión secuencial como la paralela utilizarán este valor por defecto (5) cuando el usuario no indique uno distinto.

Selección de ε.

La elección de ε es más sensible y depende de la distribución real de los datos. Para evitar que el usuario tenga que determinarlo manualmente, el sistema implementa un método automático basado en las k-distancias [3]. Para cada punto se calcula la distancia a su k-ésimo vecino más cercano (siendo k = MinPts), obteniendo así un conjunto de distancias características. A partir de ellas, el valor recomendado de ε se estima mediante la heurística:

$\varepsilon = \mu_k + \sigma_k \cdot \text{std\_scale}$

donde μ_k representa la distancia media al vecino k-ésimo, σ_k su desviación estándar asociada y std_scale es un factor ajustable por el usuario, restringido al intervalo [0 ,1], y cuyo valor por defecto será 1.0.

Para la implementación del cálculo de k-distancias se sigue el siguiente algoritmo:

Algoritmo 1

k-distance[cpu][gpu]: cálculo de la distancia al k-ésimo vecino más cercano

Requiere:

- points: conjunto de puntos
- k: índice del vecino de referencia
- N: número total de punto

Devuelve:

- k_dist: vector con la distancia al k-ésimo vecino más cercano para cada punto

```
function k-distance(points, k, N)
    Declarar k_dist [1…N]
    for i = 0 to N – 1: [PARALELO en GPU]
        Declarar min_dists [1…k]
 Inicializar min_dist a +∞
       for j = 0 to N – 1:
           if (i ≠ j):
               dist = (points[i] – points[j])²
               if (dist < min_dists[k – 1]):
                   Insertar dist en min_dists de manera ordenada
```

```
                end if
            end if
        end for
        k_dist[i] = √(min_dists[k - 1])
    end for
    return k_dist
end function
```

En CPU, ambos bucles se ejecutan de forma secuencial. En GPU, el bucle externo se paraleliza asignando un hilo por punto, mientras que el bucle interno permanece secuencial dentro de cada hilo (ILP, *Instruction-Level Parallelism*).

La complejidad computacional del algoritmo en CPU es:

$$O(N^2)$$

donde N es el número total de puntos.

En GPU, la complejidad viene dada por:

$$O\left(\frac{N \cdot N}{\text{blockDim} \cdot nSM}\right)$$

donde:

- nSM es el número de Streaming Multiprocessors de la GPU,
- blockDim=NTHREADS es el número de hilos por bloque.

Dado que:

$$\text{blockDim} \cdot nSM = ncCUDA$$

siendo $ncCUDA$ el número total de CUDA cores efectivos disponibles para ejecutar el kernel, la complejidad puede reescribirse como:

$$O\left(\frac{N^2}{ncCUDA}\right)$$

Finalmente, si se cumple que:

$$\lim_{ncCUDA \to N} O\left(\frac{N^2}{ncCUDA}\right) = O(N)$$

entonces el algoritmo tenderá a un comportamiento lineal en GPU cuando el número de núcleos disponibles sea comparable al número de puntos procesados.

3.2.2 Diseño del Grafo

El sistema representará el conjunto de datos mediante un grafo G(V, E), siguiendo la filosofía presentada en *G-DBSCAN: A GPU Accelerated Algorithm for Density-Based Clustering* [2], pero adaptando su diseño a las necesidades específicas de esta implementación. Cada punto del *dataset* se corresponde con un vértice, y se establece una arista entre dos vértices siempre que la distancia entre ellos sea menor o igual que el radio ε. Esta representación permite separar la fase de cálculo de vecindades de la fase de propagación de agregados, lo que facilita la paralelización posterior.

El grafo se almacena mediante una lista de adyacencia compacta formada por tres estructuras: un vector de grados (*degree*), un vector de índices de adyacencia (adjacent_indexes) y una lista plana de adyacencia, también conocida como lista plana de vecinos (adjacent_list). Esta representación es especialmente adecuada para GPU, ya que permite accesos contiguos y coalescentes a memoria global.

La construcción del grafo se divide en dos funciones diferenciadas:

- En la primera función, el sistema calcula cuántos vecinos tiene cada punto dentro del radio ε. Cada hilo de GPU procesa un punto distinto, evaluando su distancia a todos los demás. El resultado será el vector *degree*, donde cada posición indica el número de vecinos del punto correspondiente
- En la segunda función, una vez conocido el grado de cada punto, el sistema calcula los índices de inicio de cada lista de vecinos y rellena la estructura *adjacent_list*. Cada hilo de GPU volverá a recorrer el *dataset*, insertando los vecinos correspondientes en la posición adecuada.

Durante la construcción de este grafo, el sistema marca directamente como núcleo aquellos puntos cuyo grado sea mayor o igual que minPts, evitando así un recorrido adicional.

Algoritmo 2

build_adjacency_list [cpu,gpu]: Funcion que genera la lista de adyacencia del grafo, con los índices ya precalculados

Requiere:

- points: conjunto de puntos
- degree: vector de grados
- adjacent_indexes: índices de inicio ya calculados
- ε: radio de vecindad
- minPts: umbral para puntos núcleo
- N: número total de punto

Devuelve:

- adjacent_list: lista donde se marcan los índices de los vecinos de cada punto
- vector_type: lista que marca con un 1 si es núcleo y -1 en caso contrario

```
function build_adjacency_list(points, degree, adjacent_indexes, ε, minPts, N)
    Inicializar adjacent_list con tamaño igual a la suma de degree[i]
    Declarar vector_type [1…N]
    for i = 0 to N – 1: [PARALELO en GPU]
        offset = adjacent_indexes[i]
        for j = 0 to N – 1:
            if i ≠ j:
                dist = (points[i] – points[j])²
                if dist ≤ ε²:
                    adjacent_list[offset] = j
                    offset = offset + 1
```

```
                end if
            end if
        end for
        if degree[i] ≥ minPts:
            vector_type [i] = 1
        else
            vector_type[i] = -1
        end if
    end for
    return adjacent_list, vector_type
end function
```

Ambas funciones que intervienen en la construcción del grafo, tanto el cálculo de vecinos como la generación de la lista de adyacencia, presentan la misma complejidad computacional, ya que en ambos casos será necesario evaluar todas las distancias entre pares de puntos. Por ello, el coste total estará dominado por el doble bucle, dando lugar a una complejidad de $O(N^2)$ en CPU.

En GPU, el bucle externo se paraleliza asignando un hilo por punto, mientras que el bucle interno permanece secuencial dentro de cada hilo (ILP). Esto permite reducir el tiempo efectivo de ejecución a:

$$O\left(\frac{N^2}{ncCUDA}\right)$$

donde ncCUDA representa el número total de núcleos CUDA disponibles. En el caso ideal en el que el número de núcleos sea comparable al número de puntos, la complejidad tenderá a un comportamiento lineal, como se ha explicado en el algoritmo 1:

$$O(N)$$

3.2.3 Propagación de Agregados

Una vez construido el grafo de vecindad y clasificados los puntos núcleo, el sistema procede a la fase de propagación de agregados, cuyo objetivo es identificar los conjuntos de puntos densamente conectados que constituyen cada agrupación. Siguiendo la filosofía presentada en [2], esta fase corresponde a la segunda parte del algoritmo DBSCAN y se basa en recorrer el grafo generado en la etapa anterior para descubrir todos los puntos alcanzables desde cada punto núcleo. La estructura del grafo y la separación entre cálculo de vecindades y propagación permiten aplicar un recorrido en anchura *(Bread First Search)* altamente eficiente y altamente paralelizable en GPU, manteniendo la equivalencia exacta con la definición de densidad-conectividad del algoritmo.

Breadth-first search (BFS)

Para realizar la propagación de cada agregado, el sistema empleará un recorrido en anchura (Breadth-First Search, BFS), un algoritmo de exploración de grafos utilizado para identificar componentes conexos y procesar grafos por niveles

[13]. El BFS comienza desde un vértice inicial, en este caso, un punto núcleo, y visita primero todos sus vecinos, después los vecinos de esos vecinos, y así sucesivamente. Este proceso se gestiona mediante una estructura denominada frontera, que contiene los vértices pendientes de procesar en el nivel actual.

Funcionamiento general

El proceso seguirá los siguientes pasos:

1. Se recorren todos los puntos del *dataset*.
2. Cuando se encuentre un punto núcleo no visitado, se inicia un nuevo agregado.
3. Se ejecuta un BFS desde ese punto, añadiendo a la frontera todos los puntos vecinos.
4. Cada punto visitado durante la propagación se etiqueta con el identificador del agregado en el que nos encontramos.
5. Una vez este vacía la frontera, se selecciona el siguiente punto núcleo no visitado y se repite el proceso.

Siguiendo este proceso se garantiza que cada punto dentro de los agregados esté conectado por relaciones de densidad con sus vecinos.

Algoritmos

A continuación se presentan los algoritmos que implementan la fase de propagación de agregados mediante un recorrido en anchura (BFS) en el código. Estos algoritmos representan la traducción del diseño descrito previamente a un conjunto de pasos operativos que podrán ejecutarse tanto en CPU como en GPU. Su objetivo es identificar, a partir de los puntos núcleo, todos los puntos alcanzables dentro del grafo de vecindad, asignando a cada uno de ellos la etiqueta del agregado correspondiente.

Algoritmo 3
expand-clusters [cpu,gpu]: propagación de agregados mediante BFS. Requiere: • vector_type: lista que marca con un 1 si es núcleo y -1 en caso contrario • points: Lista de puntos • adjacent_indexes: índices de la lista de adyacencia • adjacent_list: lista de adyacencia Devuelve: • labels: vector con la etiqueta de agregado asignada a cada punto

`function expand-clusters(vector_type, points, adjacent_indexes, adjacent_list)`
`Declarar labels [1…N]`
`Inicializar labels a -1`
`cluster_id = 0`
`for i = 0 to N-1:`
`if vector_type[i] = 1 AND labels[i] = -1 then`

```
 labels[i] = cluster_id
 frontier = { i }
 while frontier ≠ ∅ do
frontier = bfs-level(frontier, labels, cluster_id, vector_type,
adjacent_indexes, adjacent_list)
 end while
 cluster_id = cluster_id + 1
end if
end for
return labels
end function
```

El Algoritmo 3 describe el proceso global de propagación de agregados. Recorre todos los puntos del *dataset* y, cada vez que encuentra un punto núcleo no visitado, inicia un nuevo agregado e inicia un recorrido BFS desde él. Durante esta propagación, todos los puntos alcanzables se etiquetan con el identificador del agregado actual. El algoritmo continúa hasta que no quedan puntos núcleo sin procesar, garantizando que cada agregado corresponda a un componente conexo del grafo de vecindad.

Algoritmo 4

bfs-level [cpu,gpu]: procesamiento de un nivel del BFS

Requiere:

- frontier: conjunto de puntos activos en el nivel actual
- labels: vector de etiquetas
- cluster_id: identificador del agregado actual
- vector_type: lista que marca con un 1 si es núcleo y -1 en caso contrario
- adjacent_indexes: índices de la lista de adyacencia
- adjacent_list: lista de adyacencia

Devuelve:

- new_frontier: conjunto de puntos que formarán el siguiente nivel

```
function bfs-level (frontier, labels, cluster_id, vector_type,
adjacent_indexes, adjacent_list)
new_frontier = {}
for cada punto p en frontier: [PARALELO en GPU]
Obtener vecinos a partir de adjacent_indexes y adjacent_list
for cada vecino v de p:
 if labels[v] = -1 then
  labels[v] = cluster_id
  if vector_type[v] = 1 then
   Añadir v a new_frontier
  end if
```

```
 end if
end for
end for
return new_frontier
end function
```

El Algoritmo 4 detalla cómo se procesa un único nivel del BFS. Para cada punto de la frontera actual, se recorren sus vecinos y se asigna la etiqueta del agregado a aquellos que aún no han sido visitados. Si alguno de estos vecinos es un punto núcleo, se añade a la nueva frontera para ser procesado en el siguiente nivel. Este procedimiento permite expandir el agregado de forma iterativa y paralelizable, avanzando nivel a nivel hasta que no queden puntos núcleo por explorar.

Análisis de Complejidad

El Algoritmo 3 presenta una complejidad computacional de $O(N^2)$ tanto en su implementación CPU como GPU. Esta complejidad cuadrática se mantiene porque, aunque la propagación interna de cada agregado se paraleliza eficientemente en GPU mediante BFS nivel por nivel, existe una limitación fundamental: los agregados deben iniciarse secuencialmente en el bucle principal.

Esta dependencia secuencial entre agregados surge de la naturaleza misma del algoritmo DBSCAN, donde cada agregado nuevo solo puede comenzar después de que se hayan identificado y propagado completamente todos los puntos del agregado anterior. Esta restricción impide paralelizar el bucle externo que recorre los puntos núcleo, manteniendo así la misma complejidad teórica en ambas arquitecturas a pesar de las optimizaciones internas de paralelización.

El Algoritmo 4 presenta una complejidad que depende del número total de puntos procesados en la frontera y del número de vecinos explorados en cada iteración. En CPU, el procesamiento es secuencial y, a lo largo de toda la propagación, cada punto y cada relación de vecindad se recorren como máximo una vez, por lo que el coste acumulado del algoritmo es $O(N + E)$, siendo N el número de puntos y E el número total de relaciones de vecindad del grafo. En GPU, este mismo trabajo se reparte entre los núcleos disponibles asignando un hilo por punto de la frontera, lo que reduce el tiempo efectivo de ejecución a $O\left(\frac{N+E}{ncCUDA}\right)$, donde ncCUDA es el número de núcleos CUDA activos. En el caso ideal en el que el número de núcleos sea comparable al número de puntos, el procesamiento de cada nivel del BFS tiende a comportarse como una operación de coste constante.

$$\lim_{ncCUDA \to N} O\left(\frac{N+E}{ncCUDA}\right) = O(1)$$

Con el análisis de complejidad completado, puede cerrarse esta fase del diseño evaluando su papel dentro del sistema global.

Resumen

La fase de propagación de agregados completa el proceso iniciado con la construcción del grafo de vecindad, permitiendo gracias a la información del grafo obtener los agregados finales. Mediante el uso de un recorrido en anchura, el sistema garantiza que cada agregado se forme a partir de todos los puntos alcanzables desde un punto núcleo, manteniendo un coste lineal respecto al tamaño del grafo. Los algoritmos presentados proporcionan una implementación clara y eficiente de este proceso, y su diseño permite aprovechar el paralelismo de la GPU para acelerar la propagación sin modificar la lógica del algoritmo.

3.3 Diseño del Módulo de Cálculo de Propiedades de los Agregados

Una vez finalizada la fase de propagación, el código dispone de un conjunto de puntos clasificados en agregados. El siguiente paso consiste en calcular un conjunto de propiedades geométricas que caracterizarán cada agrupación y que serán necesarias para su análisis posterior. A diferencia de las fases anteriores, centradas en la detección de los agregados, este módulo se encarga de obtener una descripción cuantitativa de la forma y distribución interna.

El objetivo de este módulo es obtener tres propiedades fundamentales para cada agregado: el centro de masas, el radio de giro y los valores propios del tensor de inercia.

Antes de realizar estos cálculos, el código lleva a cabo un paso previo que consiste en organizar los puntos por agregado. Este paso no modifica los datos ni introduce estructuras adicionales; simplemente agrupa los índices asociados a cada etiqueta para permitir un recorrido eficiente y evitar múltiples pasadas sobre los puntos.

En el siguiente apartado se presenta el algoritmo encargado de calcular todas las propiedades en una única función.

Algoritmo 5

compute-cluster-properties [cpu,gpu]: cálculo del centro de masas, radio de giro y valores propios del tensor de inercia

Requiere:

- points: lista de coordenadas (x, y)
- cluster_indices: índices de los puntos pertenecientes a cada agregado
- cluster_offsets: posición inicial de cada agregado en cluster_indices
- cluster_sizes: número de puntos de cada agregado
- cluster_count: número total de agregado

Devuelve:

- cluster_centers: centro de masas de cada agregado
- cluster_radii: radio de giro
- cluster_eigenvalues: valores propios del tensor de inercia
- cluster_eigenvalues_relation: relación entre valores propios

```
function compute-cluster-properties(points, cluster_indices,
```

```
                                cluster_offsets, cluster_sizes,
                                cluster_centers, cluster_radii,
                                cluster_eigenvalues,
            cluster_eigenvalues_relation,
                                cluster_count)
for cluster_id = 0 to cluster_count – 1:   [PARALELO en GPU]
    cluster_size = cluster_sizes[cluster_id]
    start_idx = cluster_offsets[cluster_id]
    // Cálculo del centro de masas (CDM)
    sum_x = 0
    sum_y = 0
    for i = 0 to cluster_size – 1:
        point_idx = cluster_indices[start_idx + i]
        sum_x = sum_x + points[2 * point_idx]
        sum_y = sum_y + points[2 * point_idx + 1]
    end for
    center_x = sum_x / cluster_size
    center_y = sum_y / cluster_size
    cluster_centers[2 * cluster_id] = center_x
    cluster_centers[2 * cluster_id + 1] = center_y
    // Radio de giro y tensor de inercia
    sum_r2 = 0
    Ixx = 0
    Iyy = 0
    Ixy = 0
    for i = 0 to cluster_size – 1:
        point_idx = cluster_indices[start_idx + i]
        dx = points[2 * point_idx]     – center_x
        dy = points[2 * point_idx + 1] – center_y
        sum_r2 = sum_r2 + (dx*dx + dy*dy)
        Ixx = Ixx + dy*dy
        Iyy = Iyy + dx*dx
        Ixy = Ixy + dx*dy
    end for
    cluster_radii[cluster_id] = sqrt(sum_r2 / cluster_size)
    Ixy = –Ixy
    // Valores propios del tensor de inercia
    trace = Ixx + Iyy
    discriminant = sqrt((Ixx – Iyy)^2 + 4 * Ixy^2 )
    lambda1 = (trace + discriminant) / 2
    lambda2 = (trace – discriminant) / 2
    cluster_eigenvalues[2 * cluster_id]     = lambda1
```

```
        cluster_eigenvalues[2 * cluster_id + 1] = lambda2
        lambda_max = max(lambda1, lambda2)
        lambda_min = min(lambda1, lambda2)
        cluster_eigenvalues_relation[cluster_id] = lambda_max / lambda_min
    end for
end function
```

El algoritmo encargado del cálculo de propiedades de los agregados procesa cada agrupación de manera independiente, recorriendo únicamente los puntos que pertenecen a ella. Para cada agregado, el procedimiento comienza obteniendo su centro de masas (CDM) mediante la suma de las coordenadas de todos sus puntos y su posterior normalización por el número de puntos del agregado. A continuación, calcula el radio de giro, que cuantifica la dispersión de los puntos respecto al CDM, acumulando las distancias cuadráticas de cada punto a dicho centro. Finalmente, el algoritmo construye el tensor de inercia bidimensional a partir de las desviaciones de los puntos respecto al CDM y obtiene sus valores propios, los cuales describen la orientación y elongación del agregado mediante una descomposición en sus ejes principales. Todo el proceso se ejecuta en una única pasada por los puntos de cada agregado, lo que permite un cálculo eficiente y altamente paralelizable en GPU.

Desde el punto de vista computacional, la complejidad del algoritmo viene determinada por el número total de agregados procesados. Dado que la ejecución en GPU asigna un hilo por agregado, el bucle externo se paraleliza completamente, de modo que cada hilo ejecuta de forma independiente el cálculo del centro de masas, el radio de giro y los valores propios del tensor de inercia para su correspondiente agrupación. Por tanto, el coste total del algoritmo está dominado por el número de agregados existentes, lo que conduce a una complejidad temporal de:

$$O(K)$$

siendo K el número de agregados detectados. En GPU, el diseño asigna un hilo por agregado, lo que permite repartir el trabajo entre los núcleos disponibles y reducir el tiempo efectivo de ejecución a:

$$O\left(\frac{K}{ncCUDA}\right)$$

donde ncCUDA representa el número de núcleos CUDA activos En el caso ideal en el que el número de núcleos sea comparable al número de agregados, el cálculo de propiedades tiende a un comportamiento cercano al tiempo constante por agregado.

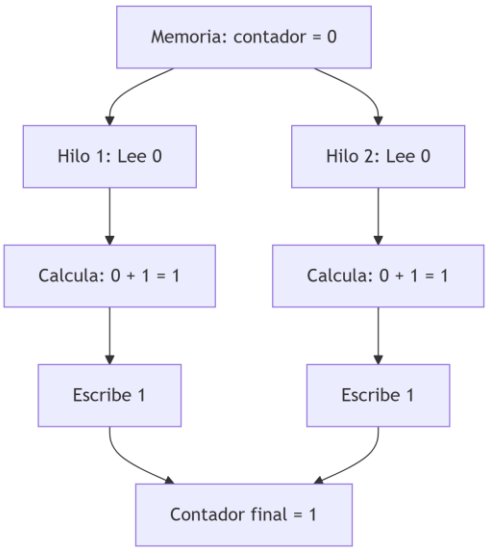
Una vez completado el cálculo de todas las propiedades asociadas a cada agregado, el módulo finaliza con la transferencia de los resultados desde la memoria de la GPU hacia la CPU. Esta operación es necesaria para su posterior utilización en las etapas de análisis o representación gráfica, ya que dichas fases se ejecutan en la CPU. Con esta transferencia se termina el proceso de caracterización geométrica de los agregados.

3.4 Paralelización: Identificación de Zonas Críticas de Memoria y Cuellos de Botella

El paso de una implementación secuencial a una paralela puede conllevar a una serie de desafíos que deben abordarse durante la fase de diseño del algoritmo. Antes iniciar la implementación en GPU, es necesario identificar y analizar las zonas críticas de memoria y los posibles cuellos de botella que pueden surgir durante la paralelización del algoritmo. Con este análisis nos aseguramos que la implementación GPU aprovecha al máximo las capacidades de la arquitectura paralela, sin provocar errores durante el proceso.

Para llevar a cabo este análisis, primero se describen los conceptos teóricos relacionados con la programación paralela, y posteriormente se realiza la identificación de posibles zonas críticas o cuellos de botella en el algoritmo.

Condiciones de Carrera



Las condiciones de carrera [8] surgen cuando los resultados de un programa dependen de la sincronización de eventos sin control, como el orden de ejecución de los hilos. Uno de los ejemplos típicos, representado en la Figura 12, se produce cuando múltiples hilos acceden a una variable compartida, en este caso para su escritura, sin sincronización. Esto provoca que varios hilos lean el mismo valor inicial y realicen la misma operación sin el valor actualizado, devolviendo un resultado incorrecto. Las operaciones atómicas y los *locks* son los principales mecanismos para evitar estas condiciones.

Figura 12: Ejemplo condición de carrera

Operaciones atómicas

Las operaciones atómicas [8] son instrucciones que garantizan la ejecución de una operación sobre un dato en memoria compartida, incluso cuando múltiples hilos intentan acceder, ya sea en lectura o escritura, concurrentemente, evitando así las condiciones de carrera. En la arquitectura CUDA algunas de las operaciones atómicas más comunes son:

- **atomicAdd():** Incremento atómico de un valor.
- **atomicSub():** Decremento atómico.
- **atomicExch():** Intercambio atómico de valores.
- **atomicCAS():** Compare-and-swap atómico.

Cuello de Botella

Los cuellos de botella [8] son componentes o procesos que limitan el rendimiento de un sistema paralelo al operar a menor velocidad que otros componentes. En GPU, los cuellos de botella suelen ser muy relevantes debido al paralelismo masivo. Unos de los más característicos es la divergencia de ejecución, donde hilos realizan diferentes operaciones provocando una reducción de la eficiencia. Otro cuello de botella relevante son las transferencias de datos entre CPU y GPU, que pueden incrementar el tiempo total si no se optimizan.

Zonas Críticas

Las zonas críticas [8] de memoria son regiones de la memoria compartida donde múltiples hilos realizan operaciones concurrentes de lectura y escritura, pudiendo generar conflictos de acceso que lleguen a disminuir el rendimiento. Estas zonas se encuentran en áreas donde hay una gran competencia de los hilos por el acceso a los mismos recursos de memoria, creando cuellos de botella.

Identificación en los Algoritmos de GPU

Una vez definidos los conceptos clave, procedemos a identificar zonas críticas y cuellos de botella en los algoritmos explicados durante este capítulo. Este análisis se ha estructurado según las fases principales del algoritmo DBSCAN.

- **Fase 1: Procesamiento de Datos de Entrada.**
  Durante esta fase se encuentra la primera zona critica del diseño. Múltiples hilos de GPU identifican píxeles no pertenecientes al fondo simultáneamente y deben escribir sus coordenadas en un array compartido. Para evitar condiciones de carrera donde varios hilos intenten escribir en la misma posición, se utilizará la operación atómica atomicAdd sobre un contador índice, garantizando que cada hilo obtenga una posición única de escritura de manera sincronizada.
- **Fase 2: Calculo del Parámetro ε.**
  En esta fase se encuentra un primer cuello de botella, ya que para el cálculo de K-Distancias, cada hilo representa un punto y tendrá que recorrer todos los puntos de manera secuencial y calcular sus distancias relativas. Esta aproximación $O(n^2)$ reduce significativamente la eficacia de la paralelización, ya que, aunque miles de hilos se ejecuten simultáneamente, cada uno debe realizar un trabajo proporcional al total de puntos, limitando la escalabilidad real del algoritmo en GPU.

- **Fase 3: Construcción del Grafo.**
  En este punto aparece una zona crítica muy similar a la de la fase 1, donde se realiza una escritura concurrente sobre un array. No obstante, en esta ocasión aplicaremos una estrategia diferente, aprovechando los índices que serán pre-calculados con anterioridad. Por otra parte, se generará un cuello de botella durante el cálculo de vecinos, ya que para cada punto es necesario recorrer el resto de puntos en busca de aquellos que cumplan las condiciones para ser considerados vecinos, del mismo modo que ocurre en la fase 2.
- **Fase 4: Propagación de Agregados (BFS Paralelo).**
  En esta fase encontramos una zona crítica que es el acceso a un *flag* compartido, pero que no requiere de atomicidad, ya que una condición

de carrera no provoca errores en el resultado final. Cada hilo puede leer y escribir el *flag* concurrentemente sin afectar la corrección del algoritmo. Por otra parte, encontramos dos cuellos de botella: la sincronización entre niveles del BFS que serializa parcialmente la propagación, y la divergencia de ejecución que ocurre cuando algunos hilos se propagan (puntos núcleo) y otros no (puntos frontera).

- **Fase 5: Cálculo de Propiedades de Agregados.**
  En esta fase nos encontramos con una zona crítica similar a la de la Fase 1, durante la organización de los puntos por agregados. Para garantizar que cada punto se asigne correctamente a su agregado sin condiciones de carrera, se utilizará la operación atómica atomicAdd sobre un contador.
  Finalmente, identificamos un cuello de botella durante las transferencias de datos entre CPU y GPU. Aunque se minimizan manteniendo los datos en memoria GPU durante el máximo tiempo posible, las transferencias finales de los resultados añaden un tiempo adicional al proceso.

Con las zonas críticas identificadas, los cuellos de botella analizados y las estrategias a seguir definidas, el diseño del algoritmo DBSCAN GPU está completo. El siguiente capítulo detalla la implementación basada en este diseño, aplicando las soluciones dadas para los posibles problemas identificados.

4 Implementación del Código

Este capítulo presenta las decisiones de implementación que se han tomado durante el desarrollo del código en sus versiones CPU y GPU. A partir del diseño descrito en el capítulo anterior, se detallan los criterios técnicos adoptados para, organizar los datos, seleccionar las bibliotecas y definir la interacción entre módulos. A continuación, se describen, las herramientas y el entorno que se han empleado, así como las decisiones más importantes y las optimizaciones realizadas.

4.1 Entorno de Desarrollo y Herramientas

La implementación del código requiere un entorno de desarrollo con las necesidades computacionales del algoritmo y con las particularidades de su ejecución. En este apartado se describen los componentes hardware y software utilizados durante el desarrollo, así como las herramientas que han permitido compilar, ejecutar y depurar las distintas versiones.

Hardware

El desarrollo y las pruebas del sistema se han realizado en un entorno que dispone de dos procesadores Intel® Xeon® Gold 6548Y+. Además, el sistema dispone de 1 TB de memoria RAM. Por otro lado, la GPU utilizada para la ejecución del algoritmo es una NVIDIA H200 NVL, basada en la arquitectura Hopper y equipada con 132 Streaming Multiprocessors (SMs) y 141 GB de memoria HBM3e.

Software

El desarrollo del sistema se ha realizado utilizando Python 3.13.5 como lenguaje principal, complementado con módulos específicos en C para aquellas partes del algoritmo que requieren un control más directo sobre la memoria y el flujo de ejecución. Sobre esta base se han empleado distintas librerías para la gestión de datos, el preprocesado y la visualización de resultados.

En particular, se han utilizado las siguientes librerías de Python:

- NumPy 2.3.5, para operaciones vectorizadas y gestión eficiente de datos en CPU.
- CuPy-CUDA12x 13.6.0, para la gestión de memoria y operaciones auxiliares en GPU, proporcionando una interfaz equivalente a NumPy sobre CUDA.
- netCDF4 1.7.3, para la lectura y manejo de ficheros científicos en formato NetCDF.
- Pillow 12.0.0, para la carga y transformación de imágenes.
- Matplotlib 3.10.7, para la generación de representaciones gráficas y la visualización de resultados.

Además, en la implementación se han empleado Numba 0.62.1 y la interfaz ctypes de Python, cuya utilización concreta se detalla en el apartado de optimizaciones, dado que su papel principal es mejorar el rendimiento de determinadas partes del código tanto en CPU como en GPU.

Compilador

Para la compilación del código CUDA se ha utilizado NVCC 12.9 (NVIDIA CUDA Compiler), correspondiente a las CUDA compilation tools, release 12.9, V12.9.86. Este compilador forma parte del ecosistema oficial de NVIDIA y garantiza compatibilidad total con la GPU H200 NVL y con las librerías CUDA empleadas en el proyecto.

4.2 Decisiones de Implementación

En este apartado se explican las decisiones de implementación adoptadas en las partes del código cuyo rendimiento resulta crítico en el algoritmo. Tal y como se estableció en el capítulo anterior, el objetivo es justificar las elecciones técnicas que afectan directamente al coste computacional del algoritmo, sin tener en cuenta aspectos como la entrada/salida, la visualización o la lógica de la aplicación. Las decisiones adoptadas son las siguientes:

- Uso de PIL y netCDF4 para unificar el preprocesado de datos:
  Para la carga de imágenes se utiliza Pillow (PIL), mientras que para ficheros científicos se emplea netCDF4. Ambas librerías nos permiten extraer los datos y convertirlos en un conjunto de puntos bidimensionales.

- Uso de NumPy en CPU:
  En la versión CPU del sistema se utiliza NumPy como libreria principal para el almacenamiento y manipulación de datos. Los puntos se representan mediante una matriz N×2, donde cada fila contiene las coordenadas bidimensionales de un punto almacenadas. Esta organización evita el uso de estructuras dinámicas de Python, que tienen

un rendimiento significativamente menor. La representación interna de NumPy, permite aprovechar de forma eficiente la jerarquía de cachés y ejecutar operaciones vectorizadas con un coste reducido.

- Uso de CuPy para la gestión de memoria y operaciones auxiliares en GPU:
  En la versión GPU se emplea CuPy, la biblioteca desarrollada por NVIDIA que ofrece una interfaz similar a NumPy. Esta elección permite gestionar los arrays directamente en la GPU, garantiza la compatibilidad con CUDA y evita transferencias de datos innecesarias. CuPy facilita la ejecución de operaciones auxiliares, como histogramas, conversiones o asignaciones, sin necesidad de escribir kernels dedicados. Gracias a ello, la implementación se puede centrar en optimizar los kernels críticos del algoritmo.

- Selección explícita de tipos de datos para la ejecución en GPU:
  Antes de transferir los datos a la GPU, la implementación convierte todas las coordenadas a float32 y se declaran las etiquetas o estados a int32. Con esto se garantiza la compatibilidad con los kernels CUDA, que están optimizados para operar con tipos de 32 bits, y evita el uso innecesario de tipos de mayor tamaño como float64 o int64; excepto en el case de los acumuladores donde se utiliza doble precisión para evitar acumulación de errores.

- Uso de distancias al cuadrado para evitar operaciones costosas:
  Todas las comparaciones de distancia se realizan utilizando valores al cuadrado, evitando así el cálculo repetido de raíces cuadradas. Esta decisión reduce de forma notable el coste computacional del algoritmo, especialmente en la fase de cálculo de vecindades, donde deben evaluarse exactamente N^2 distancias entre pares de puntos. La raíz cuadrada únicamente se calcula en las etapas donde es estrictamente necesaria, como en la obtención de las k-distancias o en el cálculo del radio de giro de los agregados, garantizando un equilibrio adecuado entre precisión y eficiencia.

- Minimización de transferencias entre CPU y GPU:
  Dado que las transferencias de datos entre CPU y GPU suponen un coste elevado, la implementación está diseñada para mantener los datos en la GPU durante toda la ejecución del algoritmo. Solo se transfieren al host los resultados estrictamente necesarios. Esta decisión reduce la latencia global del sistema y garantiza que el tiempo de ejecución esté dominado por el cálculo y no por la comunicación entre dispositivos.

- Minimización de sincronizaciones en GPU:
  En los *kernels* CUDA se ha adoptado la estrategia de reducir al mínimo las sincronizaciones explícitas entre hilos, limitando el uso de barreras únicamente a los casos estrictamente necesarios para garantizar la coherencia de los datos. Las sincronizaciones introducen tiempos de espera que afectan directamente a la ocupación efectiva de la GPU, especialmente en *kernels* con un número elevado de operaciones por hilo. Al minimizar estas barreras se evita que los hilos permanezcan inactivos, se mejora el solapamiento de operaciones y se incrementa el rendimiento global del algoritmo, particularmente en las fases de cálculo de vecindades y propagación de agregados.

- Elección del número de hilos por bloque en GPU:
  Para determinar el tamaño óptimo de bloque en la versión GPU del algoritmo, se evaluaron configuraciones de 64, 128, 256 y 512 hilos por bloque sobre distintos conjuntos de datos. Los resultados obtenidos, resumidos en la Figura 13, muestran que el tamaño de 64 hilos presenta un rendimiento inferior de forma consistente, mientras que las configuraciones de 128, 256 y 512 hilos ofrecen tiempos muy similares, con diferencias del orden de milésimas de segundo. No obstante, el tamaño de 256 hilos por bloque proporciona un equilibrio más adecuado entre ocupación, paralelismo y uso de registros en la arquitectura Hopper de la GPU NVIDIA H200 NVL, manteniendo un número elevado de bloques residentes por SM y garantizando un rendimiento estable en datasets de gran tamaño. Por este motivo, se seleccionó 256 hilos por bloque como configuración final del sistema.

Nº de puntos	64 threads	128 threads	256 threads	512 threads
57 813	1.049 s	1.043 s	1.052 s	1.044 s
93 695	1.617 s	1.600 s	1.606 s	1.608 s
107 426	1.816 s	1.796 s	1.783 s	1.781 s

Figura 13: Tiempos de ejecución para distintos tamaños de bloque en GPU.

4.3 Optimizaciones Adoptadas

Con el objetivo de mejorar el rendimiento del algoritmo DBSCAN y reducir los tiempos de ejecución obtenidos en la versión secuencial, se han llevado a cabo diversas optimizaciones tanto en CPU como en GPU. En esta sección se presentan las decisiones tomadas y el análisis comparativo de las distintas implementaciones.

4.3.1 Comparativa inicial entre CPU y GPU

Para comenzar, se evaluó el rendimiento de la versión secuencial del algoritmo frente a la versión paralela implementada en GPU. Para ello, se empleó una imagen que generaba un conjunto de 93 695 puntos, midiendo el tiempo de ejecución de cada una de las etapas del programa.

Tal como se muestra en la Figura 14, la diferencia entre ambas implementaciones es drástica. Mientras que la CPU requiere más de 4 horas para completar el proceso, la GPU necesita únicamente 1.6 segundos, logrando una aceleración superior a 10 000× en las etapas más costosas: el cálculo de ε y la construcción del grafo.

Siguiendo esta observación, se decide que es necesario optimizar la versión en CPU para obtener tiempos de ejecución que permitan un uso práctico del programa, especialmente en aquellos entornos donde no se dispone de una GPU.

93695 puntos	Tiempo extracción de puntos	Tiempo cálculo de ε	Tiempo construcción de grafo	Tiempo de propagación agregados	Tiempo obtención de propiedades	Tiempo total
CPU	0.638 s	5898.471 s (1 h 38 min 18 s)	10378.591 s (2 h 52 min 59 s)	6.858 s	0.138 s	16284.696 s (4 h 31 min 25 s)
GPU	0.186 s	0.022 s	0.031 s	1.352 s	0.016 s	1.607 s

Figura 14: Evaluación del rendimiento por etapa del DBSCAN (93 695 puntos) entre CPU y GPU.

4.3.2 Optimización en CPU mediante Numba

Tras esta primera observacion, se decidió optimizar la versión en CPU mediante Numba, un compilador JIT (Just-In-Time) que traduce funciones de Python a código máquina optimizado. Numba permite aplicar vectorización, paralelización automática y optimizaciones de bajo nivel sin necesidad de reescribir el código en C o C++, lo que lo convierte en una herramienta ideal para acelerar operaciones numéricas intensivas.

La implementación con Numba se aplicó principalmente a las fases de cálculo de ε, construcción del grafo y propagación de agregados.

93695 puntos	Tiempo extracción de puntos	Tiempo cálculo de ε	Tiempo construcción de grafo	Tiempo de propagación agregados	Tiempo obtención de propiedades	Tiempo total
CPU	0.638 s	5898.471 s (1 h 38 min 18 s)	10378.591 s (2 h 52 min 59 s)	6.858 s	0.138 s	16284.696 s (4 h 31 min 25 s)
CPU - Numba	0.655 s	10.194 s	14.381 s	0.194 s	0.141 s	25.565 s
GPU	0.186 s	0.022 s	0.031 s	1.352 s	0.016 s	1.607 s

Figura 15: Evaluación del rendimiento por etapa del DBSCAN (93 695) puntos entre CPU, CPU optimizada y GPU.

Estos resultados (Figura 15) muestran una mejoría en la versión de CPU, pasando de 4 horas 31 minutos a 25.6 segundos, lo que supone una aceleración de más de 600x.

Además, se observa que la versión en CPU optimizada con Numba obtiene un mejor rendimiento en la propagación de agregados que la versión en GPU. Esto se debe a que la implementación en GPU utiliza CuPy, lo que conlleva un coste elevado en la transferencia de datos entre Python, CuPy y la propia GPU. En esta etapa, la operación no es masivamente paralelizable y requiere lanzar un *kernel* numerosas veces, provocando un cuello de botella que penaliza el rendimiento en GPU.

4.3.3 Optimización en GPU: propagación de agregados en C

Para eliminar el cuello de botella detectado en la propagación de agregados en GPU, se decidió reescribir esta fase en C, compilarla como una librería compartida (.so) y cargarla desde Python mediante ctypes.

De esta manera reducimos la sobrecarga asociada a CuPy, permitiendo ejecutar la lógica crítica directamente en código nativo y minimizando las transferencias de datos.

93695 puntos	Tiempo extracción de puntos	Tiempo cálculo de ε	Tiempo construcción de grafo	Tiempo de propagación agregados	Tiempo obtención de propiedades	Tiempo total
CPU	0.638 s	5898.471 s (1 h 38 min 18 s)	10378.591 s (2 h 52 min 59 s)	6.858 s	0.138 s	16284.696 s (4 h 31 min 25 s)
CPU - Numba	0.655 s	10.194 s	14.381 s	0.194 s	0.141 s	25.565 s
GPU	0.186 s	0.022 s	0.031 s	1.352 s	0.016 s	1.607 s
GPU – Propagación en C	0.180 s	0.022 s	0.031 s	0.116 s	0.015 s	0.364 s

Figura 16: Evaluación del rendimiento por etapa del DBSCAN (93 695) puntos entre CPU, CPU optimizada con Numba, GPU y GPU con propagación de agregados optimizada en C.

Como se observa en la Figura 16 obtenemos una mejora notable: la propagación de agregados pasa de 1.4 segundos en la versión GPU original a 0.12 segundos en la versión optimizada en C, obteniendo incluso tiempos inferiores a los logrados por la implementación en CPU con Numba. Gracias a esta optimización, la GPU elimina por completo el cuello de botella asociado a esta fase y alcanza un rendimiento superior en la ejecución completa.

Con estas optimizaciones, el sistema alcanza un nivel de rendimiento adecuado para empezar con las pruebas finales.

5 Validación de la Implementación

La validación del sistema se ha realizado utilizando dos conjuntos de datos distintos: configuraciones generadas mediante simulaciones de dinámica molecular y fotografías reales de muestras biológicas. En este capítulo se presentarán dichos conjuntos de datos, así como los parámetros que se emplearán para la validación de los resultados.

5.1 Validación mediante simulaciones de dinámica molecular

Una parte esencial de la validación del sistema se ha realizado utilizando datos generados mediante el programa LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator). LAMMPS [14] es un código de dinámica molecular

ampliamente utilizado en modelización de materiales. Permite simular conjuntos de partículas y es capaz de representar sistemas atómicos, poliméricos, biológicos, granulares, coarse-grained o macroscópicos mediante una amplia variedad de potenciales interatómicos y condiciones de contorno. El software puede manejar sistemas bidimensionales o tridimensionales con tamaños que van desde unos pocos elementos hasta miles de millones de partículas, lo que lo convierte en una herramienta adecuada para estudiar procesos de agregación a gran escala.

En este trabajo se han empleado simulaciones proporcionadas por el cotutor, pertenecientes a estudios del grupo de Modelización y Simulación Molecular del Instituto de Química Física Blas Cabrera (IQF-CSIC). Estas simulaciones generan trayectorias con cientos de miles de partículas que presentan dinámicas de agregación bien caracterizadas.

Las configuraciones obtenidas se han procesado mediante dos enfoques complementarios:

1. Análisis directo de las coordenadas generadas por LAMMPS sobre modelos bidimensionales y aplicando el algoritmo GPU-DBSCAN sobre el conjunto resultante de puntos.

2. Conversión de una configuración especifica de la trayectoria en imagen mediante el software de visualización VMD (Visual Molecular Dynamics) [15]. Las imágenes generadas se digitalizan posteriormente para obtener un conjunto de puntos bidimensionales equivalente, permitiendo validar el comportamiento del algoritmo también a partir de representaciones visuales.

Esta doble aproximación garantiza que la implementación pueda evaluarse tanto sobre datos numéricos exactos como sobre imágenes derivadas de simulaciones, comprobando así la consistencia del sistema en distintos formatos de entrada.

5.2 Validación mediante muestras biológicas reales

La segunda vía de validación se ha llevado a cabo utilizando muestras biológicas reales, con el objetivo de evaluar el comportamiento del algoritmo en condiciones experimentales donde la variabilidad, el ruido y la complejidad morfológica son significativamente mayores que en los datos simulados. Para ello se emplearon tanto imágenes procedentes de bibliografía especializada como fotografías reales proporcionadas por el Laboratory of Neurochemistry and Cell Biology y el grupo de Statistical Physics and Complex Systems de la Federal University of Bahia (Brasil).

5.3 Parámetros de validación

Para las pruebas se emplearán las cuatro implementaciones descritas en el Capítulo 4, aplicando los siguientes criterios de evaluación:

- **Precisión en la identificación del número de agregados.**
  Se estimará la cantidad de agregados presentes en cada conjunto de datos, ya sea mediante recuento manual o utilizando un código de referencia proporcionado por el cotutor. Estos valores se compararán con los obtenidos por las distintas implementaciones del algoritmo,

permitiendo evaluar la capacidad del sistema para identificar correctamente el número de agregados presentes.

- **Análisis de propiedades morfológicas.**
  Se compararán las propiedades geométricas obtenidas por el módulo de extracción de características con los resultados generados por el código de referencia proporcionado por el cotutor. Esta comparación permite validar la consistencia de las métricas morfológicas calculadas por la implementación desarrollada.
- **Speedup.**
  El rendimiento relativo entre CPU y GPU se evaluará mediante la expresión:
  $\text{Speed-up} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}$
  donde T_{GPU} y T_{CPU} representan los tiempos de ejecución obtenidos por las versiones paralelas y secuenciales, respectivamente. El análisis se realizará comparando los dos programas implementados en GPU frente a las dos versiones equivalentes en CPU.

6 Pruebas, Resultados y Análisis

En este capítulo se presentan los resultados obtenidos tras la ejecución del conjunto de pruebas definido para evaluar el rendimiento y la validez de las distintas implementaciones del algoritmo DBSCAN desarrolladas en este trabajo.

Además, se examina el comportamiento de cada implementación en las distintas fases del algoritmo, evaluando el tiempo de ejecución y la capacidad de cada versión para escalar frente a conjuntos de datos de diferente tamaño. Finalmente, se analiza si el código desarrollado cumple con el objetivo principal del trabajo, determinando si las versiones paralelas en GPU logran una aceleración significativa respecto a la implementación en CPU.

6.1 Conjunto de Pruebas

En esta sección se describen los conjuntos de datos utilizados para la evaluación del código. Todas las pruebas se han realizado empleando un valor fijo de MinPts = 5 y un std_scale = 1, de modo que el valor de ε se ha calculado siguiendo la expresión:

$$\varepsilon = \mu_k + \sigma_k$$

Tal como se explica en el capítulo anterior, las pruebas se han llevado a cabo sobre tres tipos de datos:

1. Análisis directo de las coordenadas generadas por LAMMPS sobre modelos bidimensionales mediante un archivo en formato NetCDF.
2. Conversión de una configuración específica de la trayectoria en imagen, utilizando el software de visualización VMD.
3. Muestras biológicas reales en formato imagen, que han requerido un tratamiento previo para su correcta segmentación y análisis.

6.1.1 Imágenes utilizadas

A continuación, se presentan las imágenes empleadas en las pruebas, diferenciando entre aquellas obtenidas a partir de una configuración específica de la trayectoria simulada y la imagen correspondiente a la muestra biológica

real. Esta distinción es relevante, ya que la imagen experimental ha requerido un preprocesamiento adicional para obtener un resultado adecuado en el programa.

Imágenes obtenidas mediante la configuración especifica:

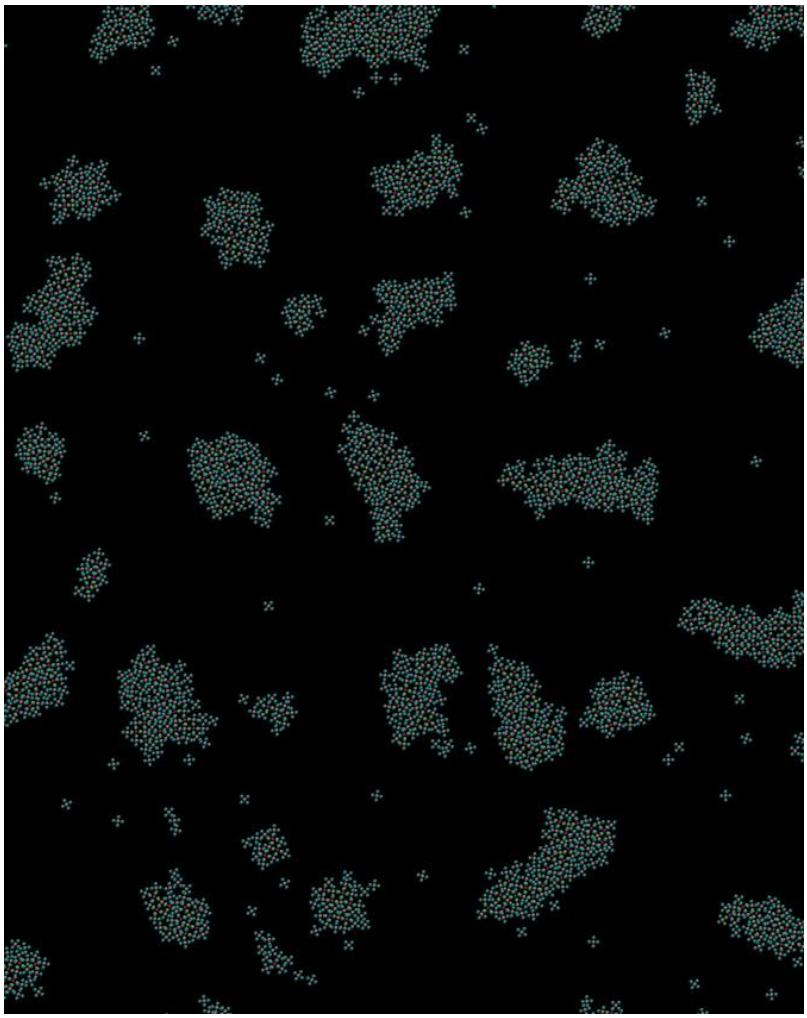


Figura 17: Imagen obtenida representando una configuración especifica de la trayectoria de un sistema de partículas anisotrópicas, con interacciones en 4 parches atractivos.

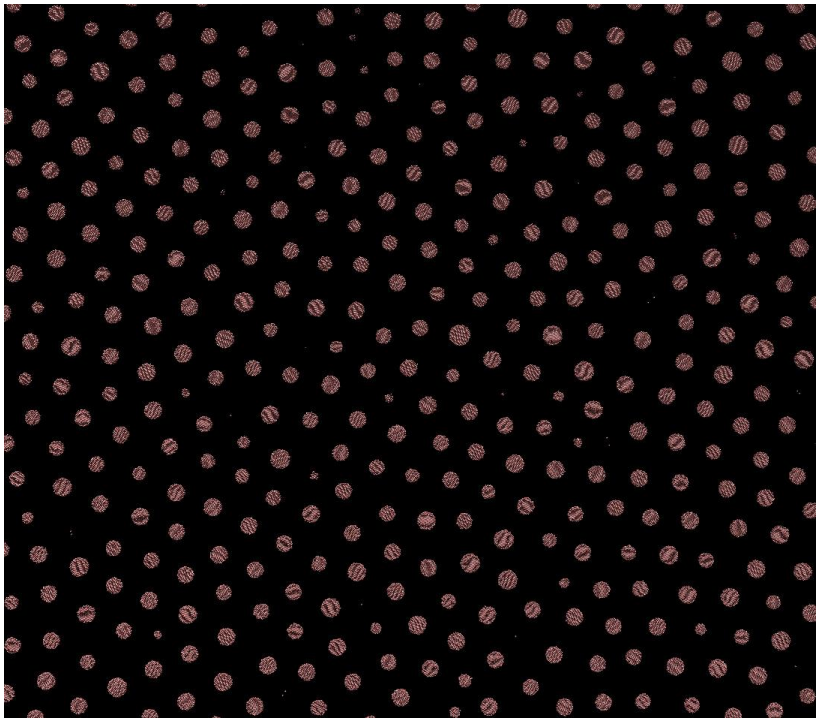


Figura 18: Imagen obtenida representando una configuración especifica de la trayectoria de agregados globulares en sistema con interacciones atractivas de corto alcance y repulsivas del largo alcance (SALR) radiales)

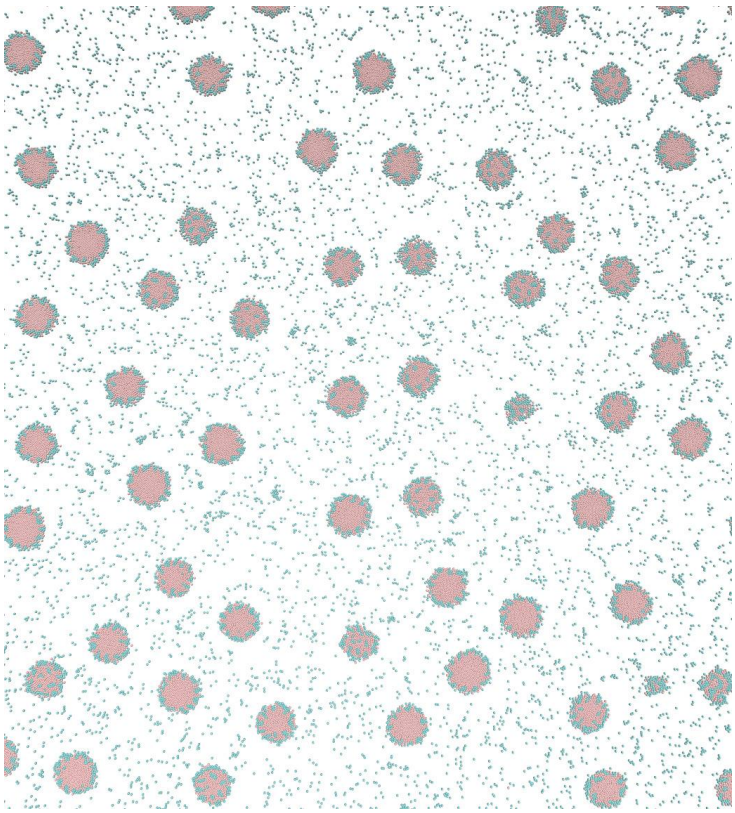


Figura 19: Imagen obtenida representando una configuración especifica de la trayectoria de una mezcla de partículas SARL.

Imagen de muestra biológica real:

Para evaluar el comportamiento del algoritmo sobre datos experimentales, se ha utilizado una fotografía de una muestra biológica procedente de un cultivo *in vitro* de neuroglioma. La imagen fue proporcionada por el Laboratory of Neurochemistry and Cell Biology y el grupo de Statistical Physics and Complex Systems de la Federal University of Bahia (Brasil), quienes facilitaron el material de microscopía utilizado en esta prueba.

En primer lugar, se parte de una imagen de microscopía de fluorescencia correspondiente a un tumor de neuroglioma cultivado in vitro (Figura 20). Esta imagen contiene la señal fluorescente asociada a las células presentes en la muestra, pero requiere un procesamiento adicional previo a la ejecución de nuestra implementación del DBSCAN para poder identificar de forma precisa las regiones celulares.

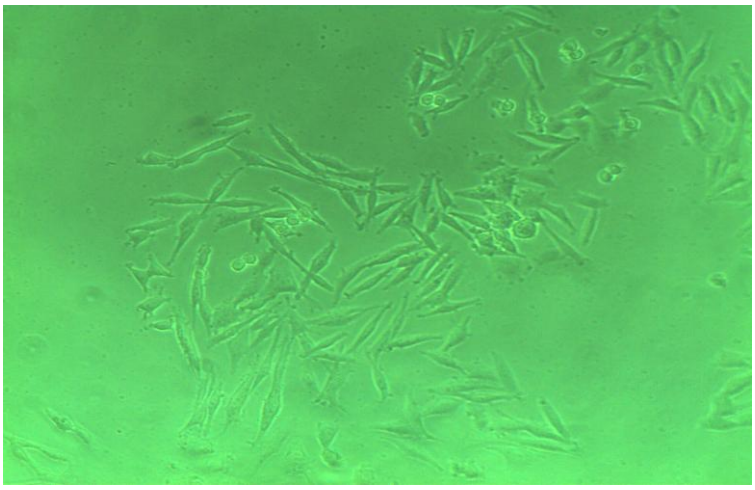


Figura 20. Imagen de microscopía de fluorescencia correspondiente a un cultivo in vitro de neuroglioma, proporcionada por el Laboratory of Neurochemistry and Cell Biology y el grupo de Statistical Physics and Complex Systems de la Federal University of Bahia (Brasil).

Para ello, se realiza la segmentación celular mediante la herramienta Cellpose-SAM [16] (Figura 21). Este procedimiento permite detectar automáticamente los contornos de las células y generar una máscara segmentada que delimita cada región de interés. El resultado es una imagen adecuada para ser utilizada como entrada en nuestro programa.

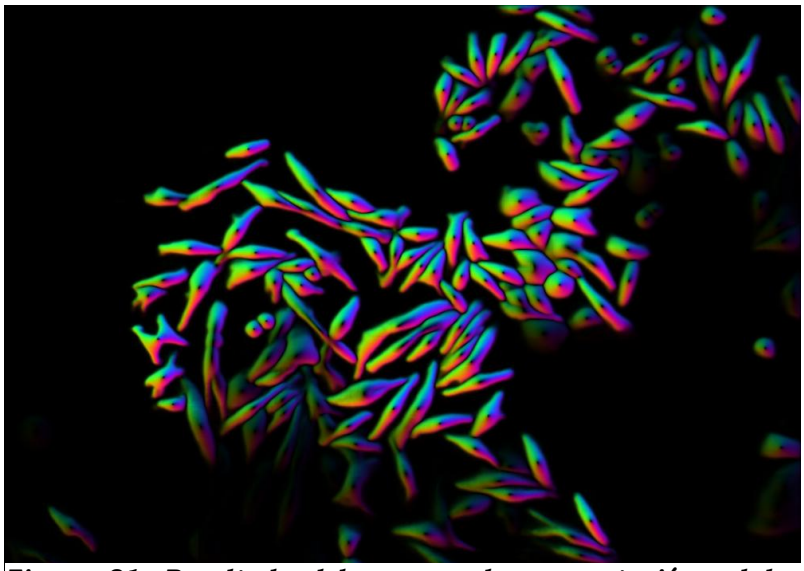


Figura 21. Resultado del proceso de segmentación celular mediante la herramienta Cellpose-SAM aplicado a la imagen de la Figura 20.

Al comparar ambas imágenes, se observa que el proceso de segmentación puede provocar la pérdida de algunas células o la aparición de regiones con intensidad muy baja, lo que podría dificultar la correcta extracción de coordenadas. No obstante, este efecto no resulta crítico para la prueba realizada, ya que la segmentación podría refinarse mediante un ajuste más cuidadoso de los parámetros o mediante un tratamiento más especializado de la imagen original. Una vez obtenida la imagen segmentada, esta constituye una representación adecuada y suficientemente estructurada para aplicar directamente la implementación del algoritmo DBSCAN desarrollada en este trabajo.

6.1.2 Archivo de coordenadas usado

Además de las imágenes, se ha empleado un archivo NetCDF procedente de una simulación molecular generada con LAMMPS para un sistema con interacciones atractivas de corto alcance y repulsivas de largo alcance como el representado en la Figura 18. Este archivo contiene las posiciones de 160 000 átomos registradas a lo largo de 501 frames temporales, junto con información adicional sobre la celda de simulación, tipos atómicos y velocidades. Para las pruebas realizadas en este trabajo se ha utilizado un único frame, correspondiente a un instante concreto de la trayectoria.

Aunque el archivo almacena las coordenadas en un espacio tridimensional, en este trabajo únicamente se emplean las dos primeras componentes (x, y).

6.2 Resultados Obtenidos

En este apartado se presentan de forma estructurada los resultados obtenidos tras la ejecución del algoritmo DBSCAN sobre los distintos conjuntos de prueba descritos previamente. Para cada imagen o *dataset* se muestran, en primer lugar, los valores fundamentales derivados del proceso de análisis, seguidos de la representación visual de los agregados y de los histogramas correspondientes a sus propiedades morfológicas.

A continuación, se incluyen las tablas de tiempos asociadas a cada implementación evaluada (CPU secuencial, CPU optimizada con Numba, GPU y GPU con propagación en C), desglosadas por fase del algoritmo. Esta presentación permite observar de manera objetiva el comportamiento del sistema en cada caso, dejando el análisis y la interpretación de los resultados para los apartados posteriores.

Resultados de las pruebas realizadas sobre las imágenes obtenidas mediante la configuración especifica:

Resultados de las pruebas de la figura 17:

Puntos	ε calculado	Número de agregados
40142	2.403	114

Figura 22. Resultados numéricos obtenidos para la imagen de la Figura 17.

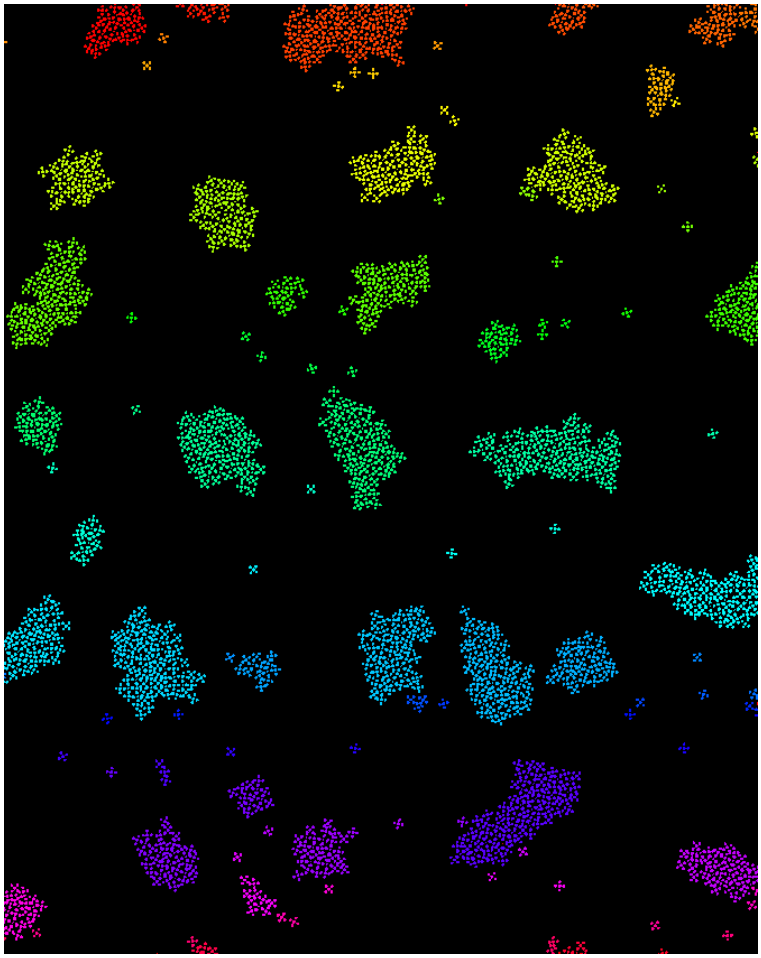


Figura 23. Imagen resultante con los agregados identificados a partir de la Figura 17.

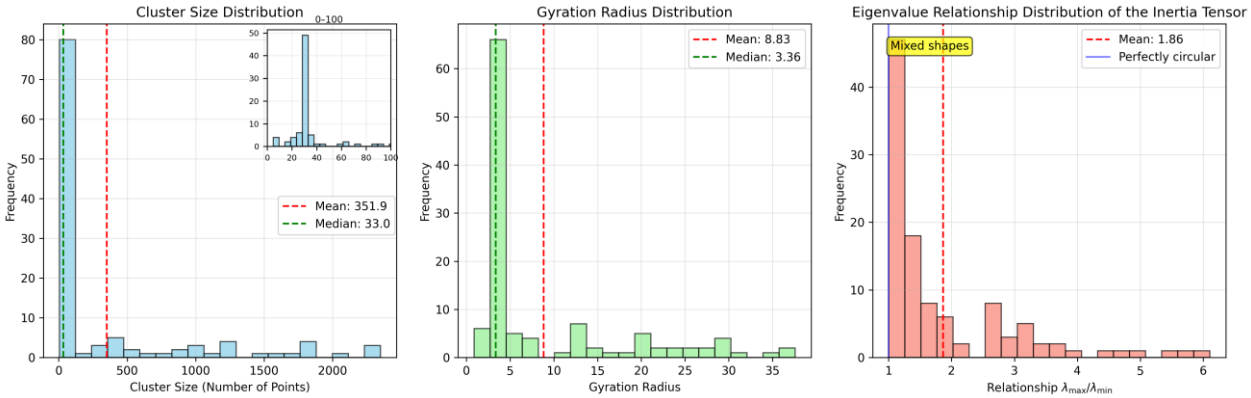


Figura 24: Histogramas de las propiedades de los agregados obtenidos a partir de la Figura 17.

	Tiempo extracción de puntos	**Tiempo cálculo de ε**	**Tiempo construcción de grafo**	**Tiempo de propagación agregados**	**Tiempo obtención de propiedades**	**Tiempo total**
CPU	0.115 s	1094.101 s (18 min 14 s)	1920.139 s (32 min)	1.691 s	0.055 s	3016.101 s (50 min 16 s)
CPU - Numba	0.114 s	2.386 s	3.908 s	0.149 s	0.058 s	6.616 s
GPU	0.175 s	0.023 s	0.029 s	0.561 s	0.015 s	0.804 s
GPU – Propagación en C	0.161 s	0.017 s	0.028 s	0.043 s	0.008 s	0.260 s

Figura 25: Tiempos de ejecución de las distintas implementaciones del algoritmo para la imagen de la Figura 17.

Resultados de las pruebas de la figura 18:

Puntos	**ε calculado**	**Número de agregados**
93695	2.337	386

Figura 26: Resultados numéricos obtenidos para la imagen de la Figura 18.

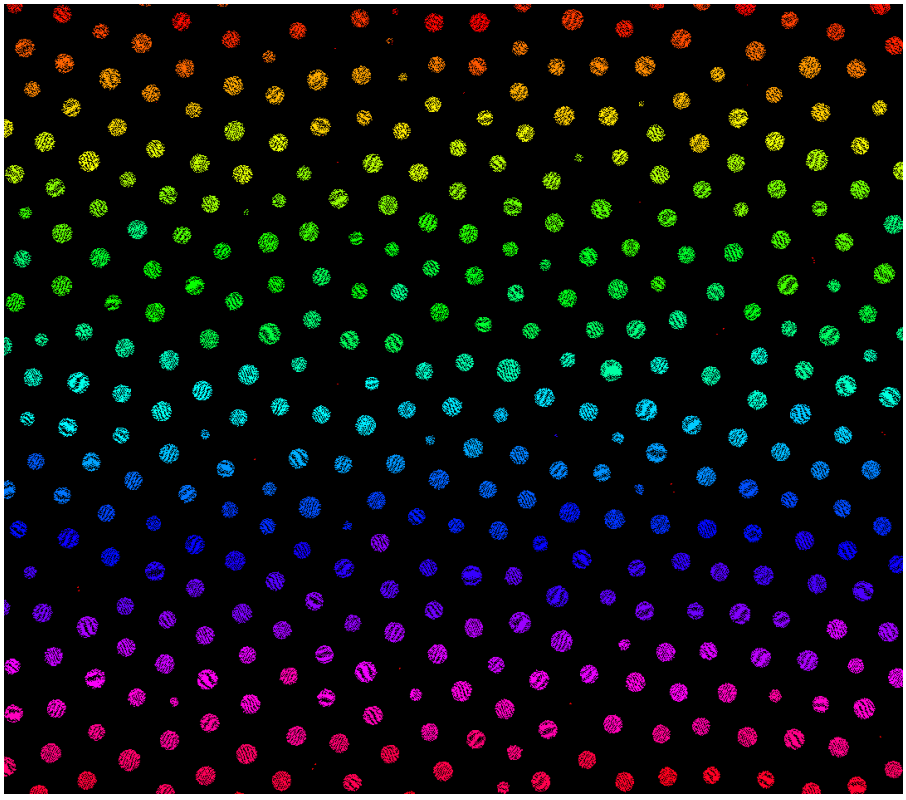


Figura 27: Imagen resultante con los agregados identificados a partir de la Figura 18.

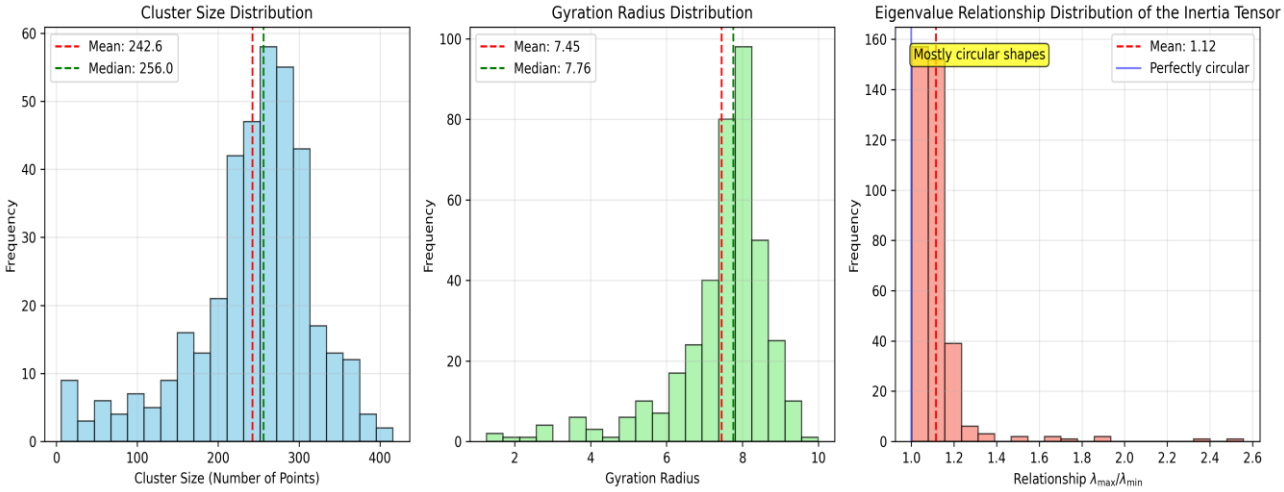


Figura 28: Histogramas de las propiedades de los agregados obtenidos a partir de la Figura 18.

	Tiempo extracción de puntos	**Tiempo cálculo de ε**	**Tiempo construcción de grafo**	**Tiempo de propagación agregados**	**Tiempo obtención de propiedades**	**Tiempo total**
CPU	0.638 s	5898.471 s (1 h	10378.591 s (2 h 52 min 59 s)	6.858 s	0.138 s	16284.696 s (4 h

		38 min 18 s)				31 min 25 s)
CPU - Numba	0.655 s	10.194 s	14.381 s	0.194 s	0.141 s	25.565 s
GPU	0.186 s	0.022 s	0.031 s	1.352 s	0.016 s	1.607 s
GPU – Propagación en C	0.180 s	0.022 s	0.031 s	0.116 s	0.015 s	0.364 s

Figura 29: Tiempos de ejecución de las distintas implementaciones del algoritmo para la imagen de la Figura 18.

Resultados de las pruebas de la figura 19:

Puntos	ε calculado	Número de agregados
176784	1.931	4565

Figura 30: Resultados numéricos obtenidos para la imagen de la Figura 19.

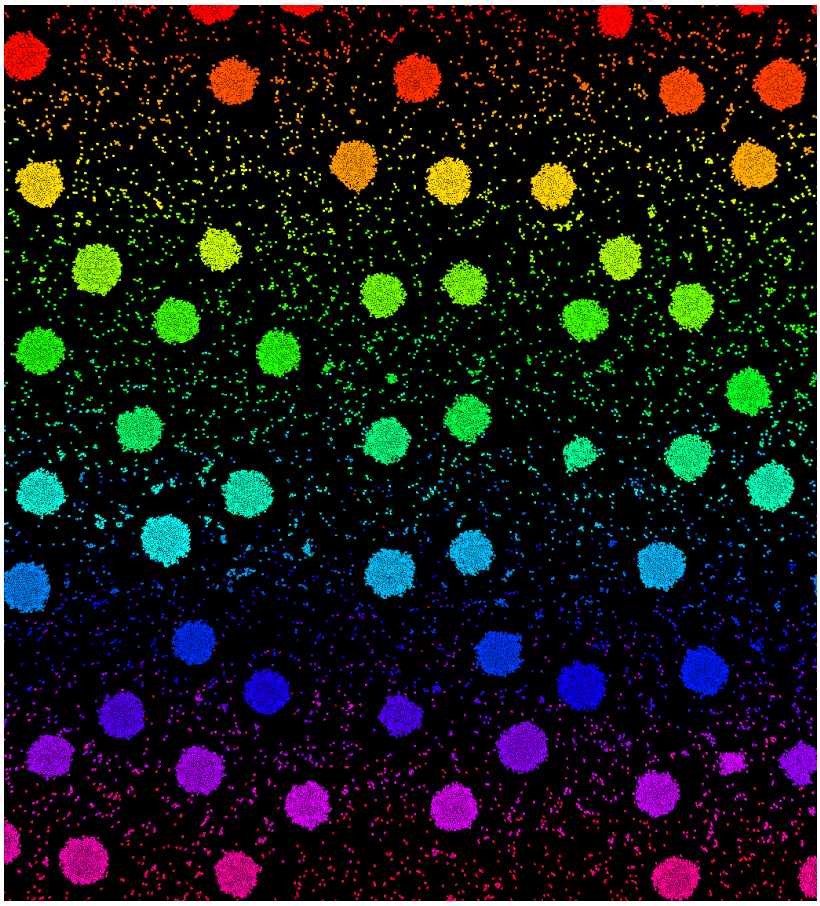


Figura 31: Imagen resultante con los agregados identificados a partir de la Figura 19.

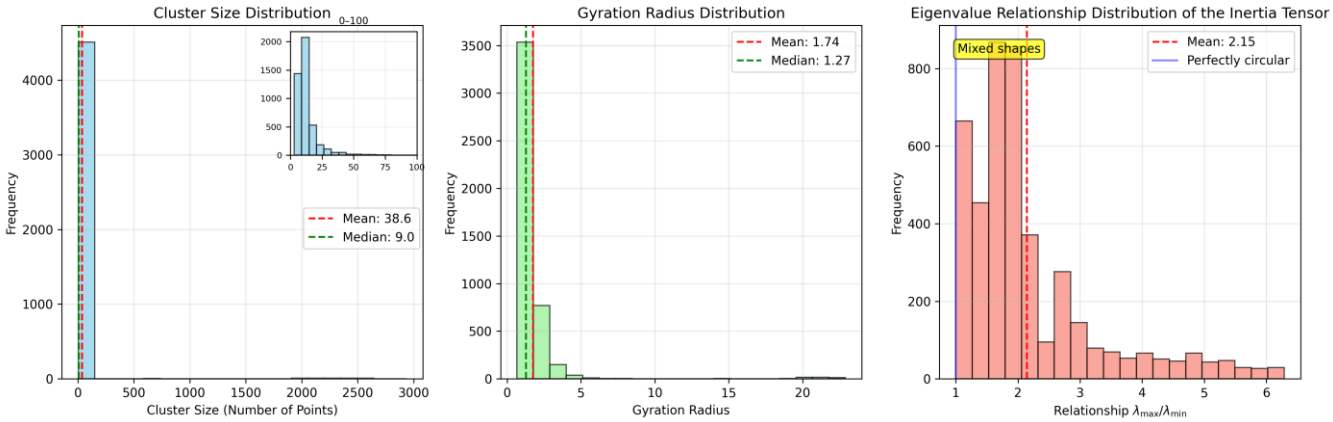


Figura 32: Histogramas de las propiedades de los agregados obtenidos a partir de la Figura 19.

	Tiempo extracción de puntos	**Tiempo cálculo de ε**	**Tiempo construcción de grafo**	**Tiempo de propagación agregados**	**Tiempo obtención de propiedades**	**Tiempo total**
CPU	0.250 s	20878.544 s (5 h 47 min 58 s)	37479.914 s (10 h 24 min 39 s)	114.388 s (1 min 54 s)	0.556 s	58473.652 s (16 h 14 min 33 s)
CPU - Numba	0.250 s	34.031 s	47.148 s	0.995 s	0.587 s	83.013 s (1 min 23 s)
GPU	0.216 s	0.049 s	0.050 s	3.274 s	0.049 s	3.639 s
GPU – Propagación en C	0.198 s	0.044 s	0.047 s	0.736 s	0.045 s	1.073 s

Figura 33: Tiempos de ejecución de las distintas implementaciones del algoritmo para la imagen de la Figura 19.

Resultados de la prueba realizada sobe la imagen de muestra biológica real:

Puntos	**ε calculado**	**Número de agregados**
107426	1.577	118

Figura 34: Resultados numéricos obtenidos para la imagen de la Figura 21.

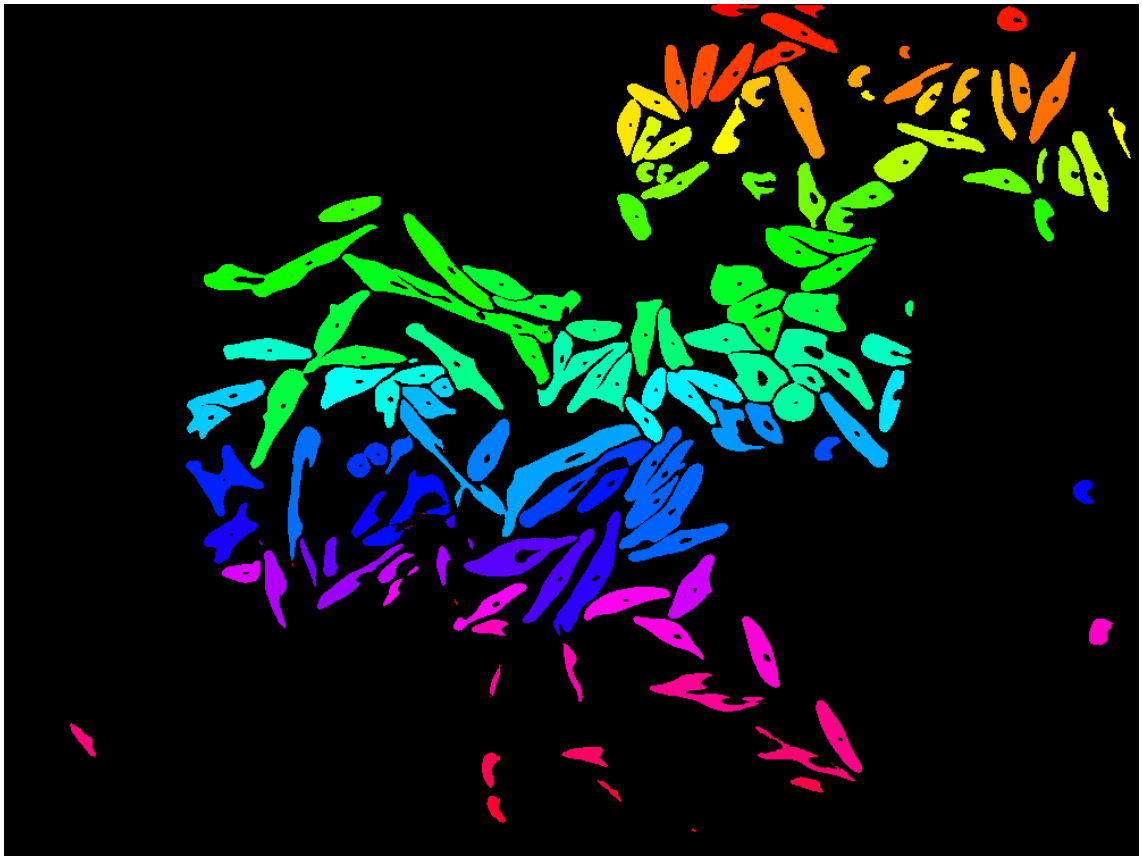


Figura 35: Imagen resultante con los agregados identificados a partir de la Figura 21.

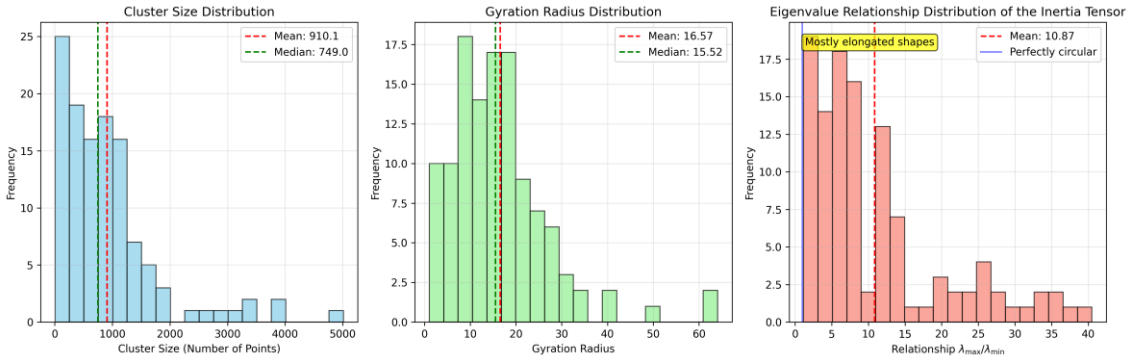


Figura 36: Histogramas de las propiedades de los agregados obtenidos a partir de la Figura 21.

	Tiempo extracción de puntos	**Tiempo cálculo de ε**	**Tiempo construcción de grafo**	**Tiempo de propagación agregados**	**Tiempo obtención de propiedades**	**Tiempo total**
CPU	0.164 s	7683.405 s (2 h 8 min 3 s)	13719.833 s (3 h 48 min 39 s)	6.624 s	0.143 s	21410.169 s (5 h 56 min 50 s)

CPU - Numba	0.164 s	12.241 s	18.201 s	0.187 s	0.152 s	30.946 s
GPU	0.221 s	0.026 s	0.037 s	1.549 s	0.016 s	1.850 s
GPU – Propagación en C	0.169 s	0.021 s	0.034 s	0.127 s	0.007 s	0.360 s

Figura 37: Tiempos de ejecución de las distintas implementaciones del algoritmo para la imagen de la Figura 21.

Resultados de la prueba realizada sobre el archivo de coordenadas:

Puntos	ε calculado	Número de agregados
160000	10.746	542

Figura 38: Resultados numéricos obtenidos para el archivo de coordenadas.

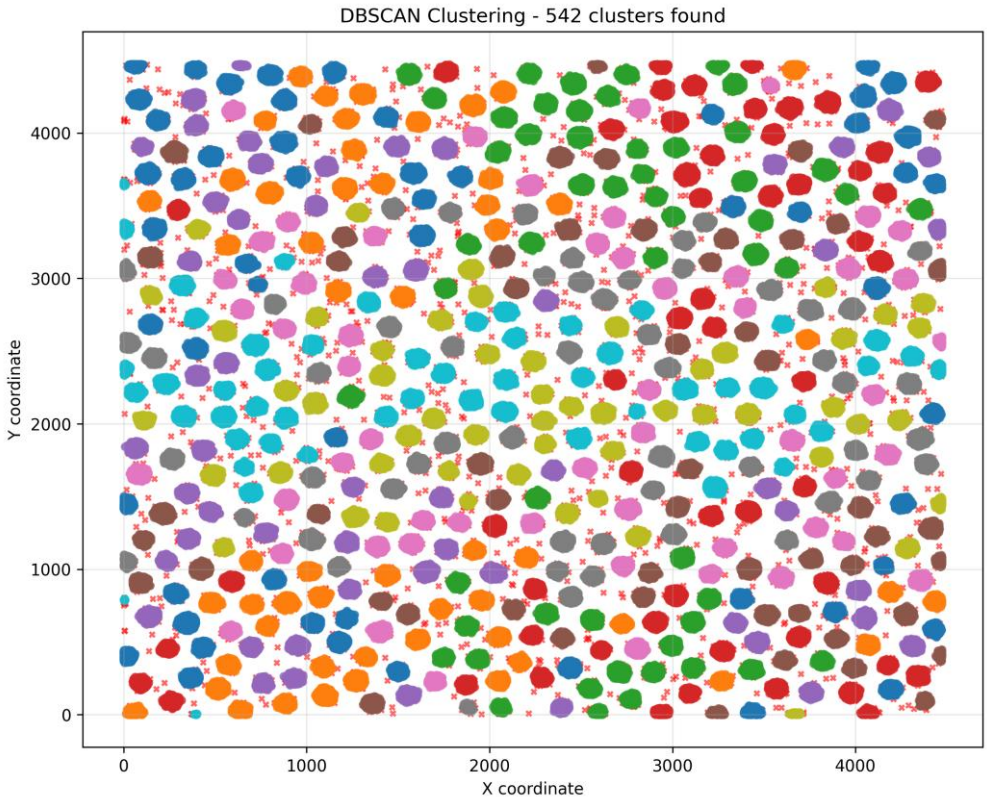


Figura 39: Imagen resultante con los agregados identificados a partir del archivo de coordenadas.

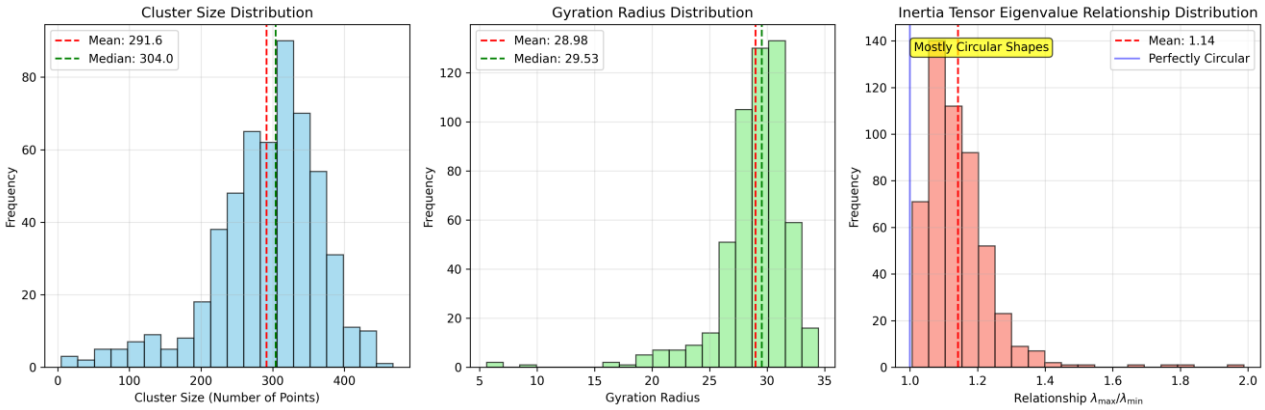


Figura 36: Histogramas de las propiedades de los agregados obtenidos a partir del archivo de coordenadas.

	Tiempo extracción de puntos	**Tiempo cálculo de ε**	**Tiempo construcción de grafo**	**Tiempo de propagación agregados**	**Tiempo obtención de propiedades**	**Tiempo total**
CPU	0.435 s	16167.818 s (4 h 29 min 27 s)	30103.959 s (8 h 21 min 43 s)	21.450 s	0.245 s	46292.948 s (12 h 51 min 32 s)
CPU - Numba	0.722 s	23.420 s	38.973 s	0.290 s	0.247 s	63.654 s (1 min 3 s)
GPU	0.558 s	0.037 s	0.045 s	2.199 s	0.017 s	2.859 s
GPU – Propagación en C	0.542 s	0.033 s	0.044 s	0.169 s	0.013 s	0.802 s

Figura 37: Tiempos de ejecución de las distintas implementaciones del algoritmo para el archivo de coordenadas.

6.3 Análisis de Resultados

Una vez presentados los resultados obtenidos en las simulaciones de las pruebas, este apartado se centra en su análisis. El objetivo es evaluar tanto el correcto funcionamiento del algoritmo como el coste computacional.

6.3.1 Validación de la Detección de Agregados

Para comprobar el funcionamiento del algoritmo implementado en este trabajo, se ha realizado una validación utilizando el archivo de coordenadas en formato NetCDF. Este conjunto de datos nos permite evaluar el algoritmo directamente sobre coordenadas ya definidas, evitando así, el posible margen de error asociado al proceso de digitalización de imágenes. De este modo, esta validación se centra en la lógica del algoritmo y no en las etapas previas de preprocesamiento de imágenes.

El archivo NetCDF se ha procesado tanto con el código TRJ_ANALYSIS desarrollado por el cotutor en CUDA-Fortran para análisis de trayectorias de

Dinámica Molecular, así como con la implementación desarrollada en este trabajo, empleando los mismos parámetros en ambos casos. La comparación de los resultados obtenidos permite verificar si los valores generados tras la simulación son correctos.

Los parámetros utilizados y los resultados obtenidos y esperados han sido los siguientes:

- Epsilon: 10.746 (valor calculado por nuestro programa y usado en ambas ejecuciones)
- MinPts: 5
- Número de agregados esperados: 542
- Número de agregados obtenidos en todas las implementaciones: 542

Al haber obtenido el mismo número exacto de agregados en todas las implementaciones, así como en el código de referencia proporcionado, se confirma que el programa ejecuta correctamente el algoritmo DBSCAN y que las optimizaciones introducidas no alteran el proceso de agrupamiento.

La Figura 38 muestra la distribución de tamaños de clúster obtenida mediante el programa de análisis proporcionado por el cotutor. Al comparar esta distribución con la generada durante las simulaciones del código implementado en este trabajo, se observa que ambas presentan un patrón prácticamente idéntico. De este modo, no solo se valida que la detección de agregados es correcta, sino también que sus características se reproducen sin ningún tipo de error.

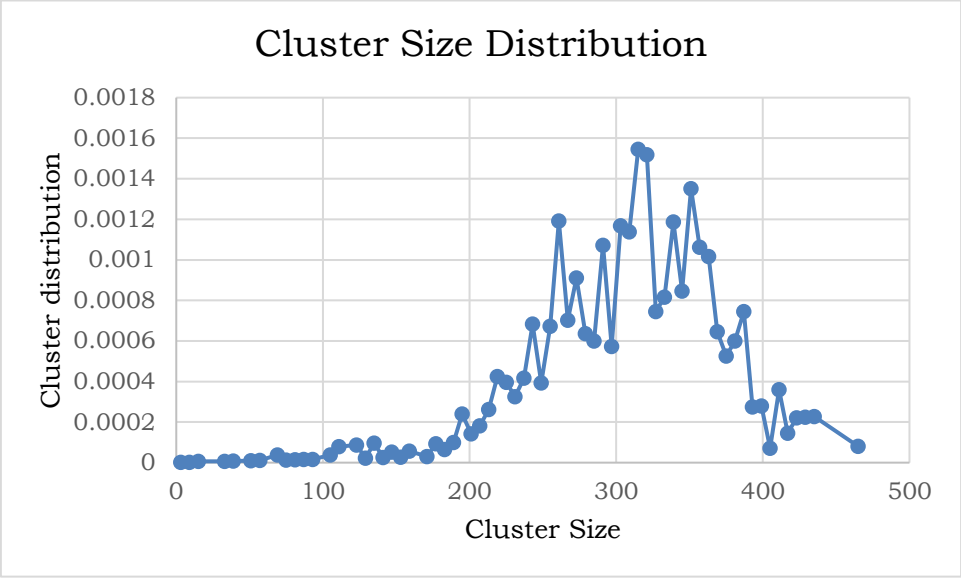


Figura 38: Distribución de tamaños de agregados obtenida a partir del código TRJ_ANALYSIS proporcionado por el cotutor.

6.3.2 Análisis de tiempos totales

El análisis de tiempos totales permite evaluar el impacto de cada una de las optimizaciones introducidas en la implementación del algoritmo. La Tabla de la Figura 39 recoge los tiempos de ejecución obtenidos para las cuatro variantes evaluadas aplicadas a los cinco conjuntos de datos con tamaños comprendidos entre 40 142 y 176 784 puntos presentados anteriormente. Estos valores constituyen la base para el estudio comparativo del rendimiento y para la interpretación de las gráficas posteriores.

Figura/Archivo	Figura 17	Figura 18	Figura 21	Archivo de coordenadas	Figura 19
Puntos	40142	93695	107426	160000	176784
Tiempo total CPU (segundos)	3016.101	16284.696	21410.169	46292.948	58473.652
Tiempo total CPU – Numba (segundos)	6.616	25.565	30.946	63.654	83.013
Tiempo total GPU (segundos)	0.804	1.607	1.85	2.859	3.639
Tiempo total GPU - Propagación en C (segundos)	0.26	0.364	0.36	0.802	1.073

Figura 39: Tiempos de ejecución obtenidos para las distintas implementaciones del algoritmo DBSCAN en función del número de puntos procesados en cada imagen o archivo de coordenadas.

Los resultados muestran diferencias extremadamente significativas entre las distintas implementaciones. La versión secuencial en CPU presenta tiempos de ejecución muy elevados, que oscilan entre aproximadamente 3 000 y 58 000 segundos. Este comportamiento confirma que la implementación sin optimizaciones no es viable para procesar imágenes de tamaño medio o grande, ya que el coste computacional crece rápidamente con el número de puntos.

La optimización mediante Numba reduce los tiempos de ejecución, situándolos en un rango de 6 a 83 segundos. Esta mejora de varios órdenes de magnitud se debe a la compilación JIT y a la vectorización automática de las operaciones más costosas. La gráfica en escala logarítmica (Figura 40) permite visualizar esta diferencia, mostrando cómo la curva de Numba se separa de la CPU de forma consistente a lo largo de todos los tamaños de entrada.

La implementación en GPU introduce una aceleración muy notable. Los tiempos se reducen a valores comprendidos entre 0.8 y 3.6 segundos, lo que supone una mejora sustancial respecto a la versión con Numba. Este comportamiento confirma que las fases paralelizables del algoritmo, especialmente la construcción del grafo de vecindad, se benefician de manera directa de la arquitectura masivamente paralela de la GPU.

Finalmente, la versión GPU con propagación en C alcanza los mejores resultados de todas las variantes evaluadas. Los tiempos obtenidos se sitúan

entre 0.26 y 1.07 segundos, lo que representa una mejora adicional respecto a la GPU pura. Esta reducción se debe a la optimización específica de la fase de propagación de agregados, que elimina sobrecostes asociados a la gestión de memoria y a la recursividad implícita del algoritmo. La curva correspondiente en la Figura 40 es la que presenta menor pendiente y menor orden de magnitud, evidenciando un comportamiento altamente eficiente incluso para los conjuntos de datos más grandes.

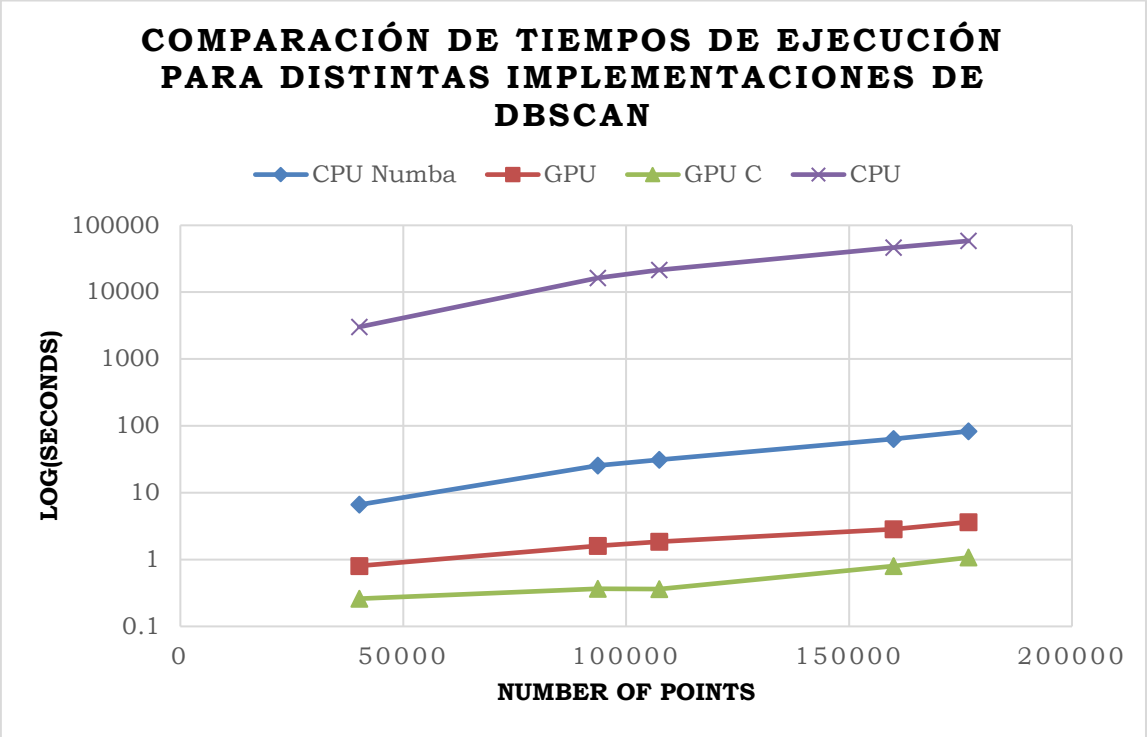


Figura 40: Comparación de los tiempos de ejecución de las distintas implementaciones del algoritmo DBSCAN en función del número de puntos procesados. Representación semi-log para mejorar la visibilidad.

Implementación	Ecuación de la recta
CPU	y = 0.4075x - 18021
CPU – Numba	y = 0.0006x - 21.778
GPU	y = 2E-05x - 0.1545
GPU - Propagación en C	y = 6E-06x - 0.1118

Figura 41: Ecuaciones de las rectas de regresión lineal correspondientes a cada implementación del algoritmo DBSCAN, obtenidas a partir de los tiempos de ejecución en función del número de puntos procesados.

La Figura 41 recoge las ecuaciones de las rectas de regresión lineal (sobre los valores en bruto, sin aplicar transformaciones logarítmicas, a diferencia de los representados en la Figura 40 (semi-log)) ajustadas a los tiempos de ejecución en función del número de puntos. El análisis de las pendientes permite cuantificar de forma explícita la eficiencia relativa de cada implementación. La versión secuencial en CPU presenta una pendiente de aproximadamente 0.4075, lo que indica que el tiempo crece de forma muy pronunciada con el tamaño del conjunto de datos. En contraste, la pendiente de la versión con Numba se reduce a 0.0006, lo que supone una disminución de más de tres órdenes de

magnitud respecto a la CPU secuencial. Las implementaciones en GPU muestran pendientes aún menores: 2×10^{-5} para la GPU pura y 6×10^{-6} para la GPU con propagación en C. Estas pendientes extremadamente bajas reflejan que el tiempo de ejecución apenas aumenta al incrementar el número de puntos, lo que confirma la escalabilidad y eficiencia de las versiones paralelas.

En conjunto, el análisis de tiempos totales confirma que las optimizaciones introducidas permiten transformar un algoritmo inicialmente prohibitivo en términos computacionales en una solución capaz de procesar imágenes de gran tamaño en segundos o incluso fracciones de segundo. En el apartado siguientes se analizarán los *speed-ups* relativos entre implementaciones, lo que permitirá cuantificar de forma explícita la magnitud de estas mejoras.

6.3.3 Análisis de *Speed-up*

El análisis de *speed-up* permite cuantificar de forma directa la mejora relativa obtenida por cada implementación del algoritmo DBSCAN respecto a las demás. A diferencia del análisis de tiempos absolutos, que muestra el coste computacional en segundos, el *speed-up* expresa cuántas veces una versión es más rápida que otra, proporcionando una medida normalizada y especialmente útil para comparar arquitecturas heterogéneas. La Figura 42 recoge los *speed-ups* calculados para cuatro combinaciones relevantes: CPU frente a GPU, CPU frente a GPU con propagación en C, CPU con Numba frente a GPU y CPU con Numba frente a GPU con propagación en C.

Los resultados muestran que las diferencias entre implementaciones son extremadamente significativas. El *speed-up* entre la CPU secuencial y la GPU alcanza valores comprendidos entre aproximadamente 3 700 y 16 000, lo que indica que la versión paralela es miles de veces más rápida que la versión secuencial incluso para los conjuntos de datos más pequeños. Esta tendencia se acentúa aún más en la comparación entre CPU secuencial y GPU con propagación en C, donde los speed-ups oscilan entre 11 600 y casi 60 000. Estos valores reflejan la magnitud del impacto que tiene la paralelización masiva combinada con optimizaciones específicas en la fase de propagación de agregados.

En el caso de las comparaciones que toman como referencia la CPU optimizada con Numba, los *speed-ups* son más moderados, pero siguen mostrando mejoras muy significativas. La GPU supera a la versión con Numba por factores que van desde 8 hasta 22, mientras que la GPU con propagación en C alcanza valores entre 25 y 86. Estos resultados confirman que, incluso tras aplicar optimizaciones JIT y vectorización en CPU, la arquitectura GPU continúa ofreciendo ventajas sustanciales, especialmente en las fases del algoritmo con mayor grado de paralelismo.

Figura/Archivo	Figura 17	Figura 18	Figura 21	Archivo de coordenadas	Figura 19
Puntos	40142	93695	107426	160000	176784
Speed-up (CPU / GPU)	3751.369	10133.600	11573.064	16192.007	16068.604
Speed-up (CPU / GPU –	11600.388	44738.175	59472.691	57721.880	54495.481

Propagación en C)					
Speed-up (CPU – Numba / GPU)	8.229	15.909	16.728	22.264	22.812
Speed-up (CPU – Numba / GPU – Propagación en C)	25.446	70.234	85.961	79.369	77.365

Figura 42: Speed-ups relativos calculados a partir de los tiempos de ejecución de las distintas implementaciones del algoritmo DBSCAN.

La Figura 43 muestra la evolución de estos *speed-ups* en función del número de puntos procesados. Aunque los valores presentan cierta variabilidad entre conjuntos de datos, las tendencias generales son claras: los *speed-ups* más elevados corresponden sistemáticamente a la comparación entre CPU secuencial y GPU con propagación en C, mientras que los menores se observan en la comparación entre CPU con Numba y GPU. En todos los casos, los factores de aceleración se mantienen en rangos muy elevados, lo que evidencia la escalabilidad de las implementaciones paralelas.

Para complementar este análisis, la Figura 44 presenta las ecuaciones de las rectas (sobre los valores en bruto, sin aplicar transformaciones logarítmicas, a diferencia de los representados en la Figura 43 (semi-log))) de regresión lineal ajustadas a cada tipo de speed-up. La pendiente de cada recta permite caracterizar la evolución del factor de aceleración a medida que aumenta el tamaño del conjunto de datos. La comparación entre CPU secuencial y GPU con propagación en C presenta la pendiente más elevada (0.2972), lo que indica que el speed-up crece de forma más pronunciada con el tamaño del problema. En contraste, las comparaciones basadas en CPU con Numba muestran pendientes mucho menores (0.0001 y 0.0003), lo que refleja que, aunque la GPU sigue siendo más eficiente, la diferencia relativa crece de forma más moderada cuando la referencia ya incorpora optimizaciones en CPU.

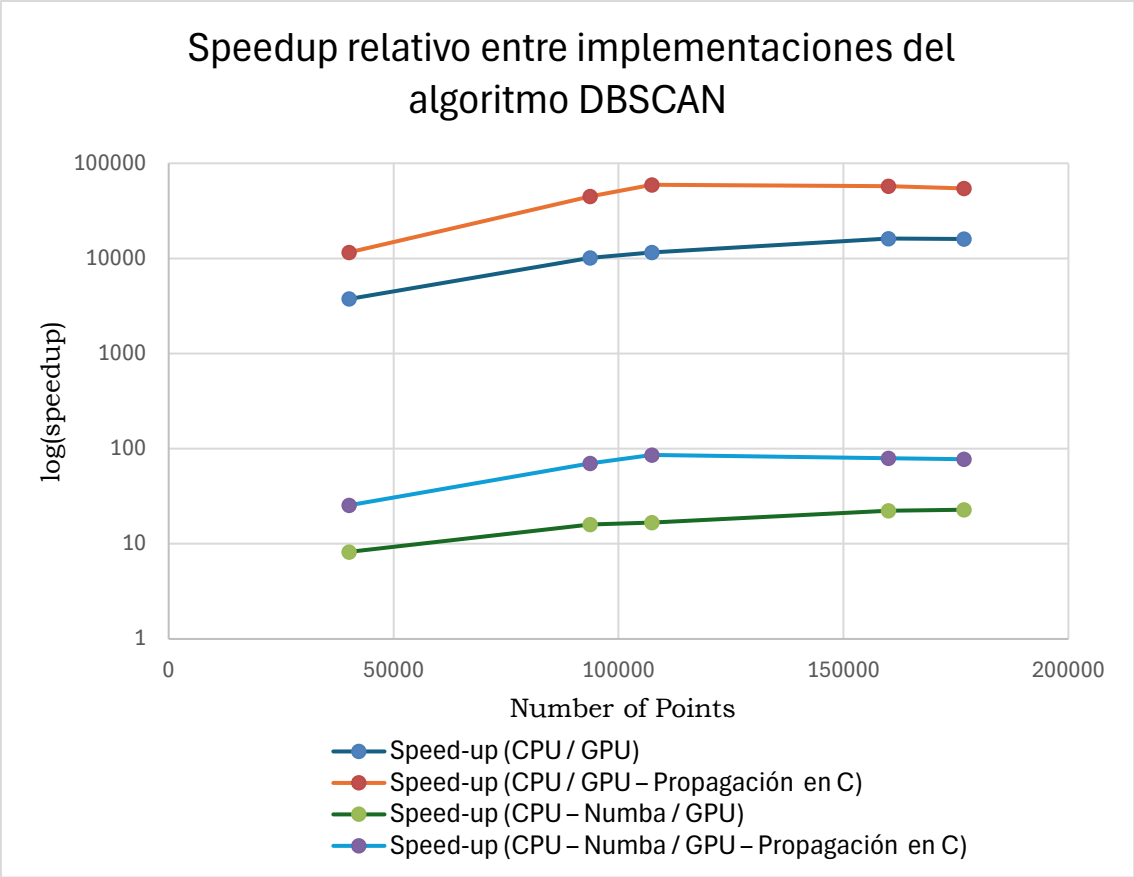


Figura 43: Representación de los factores de speedup entre CPU, CPU con Numba, GPU y GPU con propagación en C en función del tamaño del conjunto de datos. Representación semi-log para mejorar la visibilidad.

Tipo speed-up	**Ecuación de la recta**
Speed-up (CPU / GPU)	y = 0.0922x + 888.5
Speed-up (CPU / GPU – Propagación en C)	y = 0.2972x + 11243
Speed-up (CPU – Numba / GPU)	y = 0.0001x + 4.8365
Speed-up (CPU – Numba / GPU – Propagación en C)	y = 0.0003x + 28.095

Figura 44: Ecuaciones de las rectas de regresión lineal asociadas a cada tipo de speedup.

En conjunto, el análisis del *speed-up* confirma que las implementaciones en GPU, especialmente la versión con propagación en C, proporcionan mejoras de rendimiento respecto a las versiones en CPU. Estos resultados no solo muestran la eficacia de las optimizaciones, sino que demuestran que el trabajo desarrollado cumple el objetivo planteado al inicio del proyecto: obtener una implementación del DBSCAN en GPU valida y capaz de procesar conjuntos de datos de gran tamaño en tiempos inferiores a los alcanzados por las versiones en CPU.

7 Conclusiones

Este capítulo recoge las conclusiones más destacadas del desarrollo de este trabajo, destacando los problemas identificados en la implementación y validación del código, y planteando potenciales mejoras que podrían abordarse en un futuro.

7.1 Conclusiones generales

El desarrollo de este trabajo ha permitido completar una implementación funcional del algoritmo DBSCAN en GPU para análisis de agregados sobre conjuntos de datos bidimensionales e imágenes, comparando su rendimiento con la correspondiente versión secuencial en CPU. Los resultados obtenidos muestran que la paralelización en GPU aporta mejoras de muy notables rendimiento incluso con versiones optimizadas del código CPU (con vectorización y paralelización automáticas). Se ha comprobado además comportamiento correcto del algoritmo paralelo, mediante diversas pruebas de validación. El código es versátil y fácilmente ampliable a otra dimensionalidad o para incorporar la extracción conjuntos alternativos de características. Los objetivos planteados han sido alcanzados satisfactoriamente.

7.2 Análisis de problemas y posibles mejoras

Durante el desarrollo de este proyecto se han podido identificar diversos problemas, todos relacionados con la calidad de las imágenes utilizadas como entrada. En algunos casos la baja resolución de estas o contrastes insuficientes pueden dificultar la identificación de los puntos antes de la ejecución del algoritmo, lo que puede afectar a la detección de agregados y por lo tanto, obtener resultados incorrectos o inconcluyentes.

Como posibles mejoras, una primera opción sería incorporar un módulo de preprocesado de imágenes con el fin de mejorar la calidad de los datos antes de su análisis y así evitar los problemas mencionados anteriormente. Asimismo, los resultados podrían complementarse con modelos basados en redes neuronales para una identificación más robusta de formas y estructuras específicas. Por último, desde el punto de vista del rendimiento, la implementación de estructuras espaciales como *linked cells* [17] permitiría determinar los vecinos de cada punto de forma más eficiente mejorando el rendimiento global del algoritmo, reduciendo la complejidad de esa etapa del cálculo de $O(N^2)$ a $O(N)$, lo que sería esencial para $N>10^6$.

8 Análisis de Impacto

En este capítulo se examina el impacto potencial de los resultados obtenidos en este Trabajo de Fin de Grado en los ámbitos personal, empresarial, social, económico, medioambiental y cultural. Asimismo, se incluye una relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS) y se describen las decisiones tomadas durante el desarrollo del trabajo teniendo en cuenta estos aspectos.

8.1 Impacto Personal

Desde una perspectiva personal, el proyecto me ha permitido adquirir competencias relacionadas con la computación paralela en GPU, la optimización de algoritmos y me he familiarizado con arquitectura CUDA. Gracias a este trabajo he aprendido a programar en GPU utilizando Python, además de ampliar mis conocimientos en Python y en C, y mejorar la gestión de memoria durante el desarrollo de programas.

8.2 Impacto Empresarial

En el ámbito empresarial, la implementación de un algoritmo DBSCAN en GPU puede resultar muy relevante en sectores donde el análisis de grandes volúmenes de datos es una actividad común, como la biotecnología o la industria farmacéutica. La reducción del tiempo de ejecución permite disminuir los costes operativos y mejorar la capacidad de respuesta ante tareas que requieren análisis repetitivos.

8.3 Impacto Social y Cultural

Desde una perspectiva social y cultural, la aceleración de un algoritmo de agregación sobre imágenes o datos puede contribuir a mejorar procesos en ámbitos como la investigación biomédica, la detección de anomalías en imágenes clínicas o la monitorización de procesos biológicos, tal y como se ha demostrado en una de las pruebas realizadas. La capacidad de procesar grandes volúmenes de datos en menos tiempo facilita la realización de estudios, lo que puede repercutir de manera positiva en la calidad de los diagnósticos y en la toma de decisiones informadas.

8.4 Impacto Económico y Medioambiental

Desde el punto de vista económico, la aceleración obtenida permite disminuir el coste computacional del proceso en grandes volúmenes de datos. En entornos donde el tiempo de cálculo es un recurso importante esta reducción puede suponer una mejora de la productividad y una optimización del uso de infraestructura.

En el plano medioambiental, el consumo energético es un factor relevante. Aunque las GPUs presentan una potencia superior a las CPUs, un tiempo de ejecución menor permite reducir la energía consumida. Esta relación se muestra en la siguiente figura, donde se compara el consumo energético estimado de las distintas versiones del algoritmo:

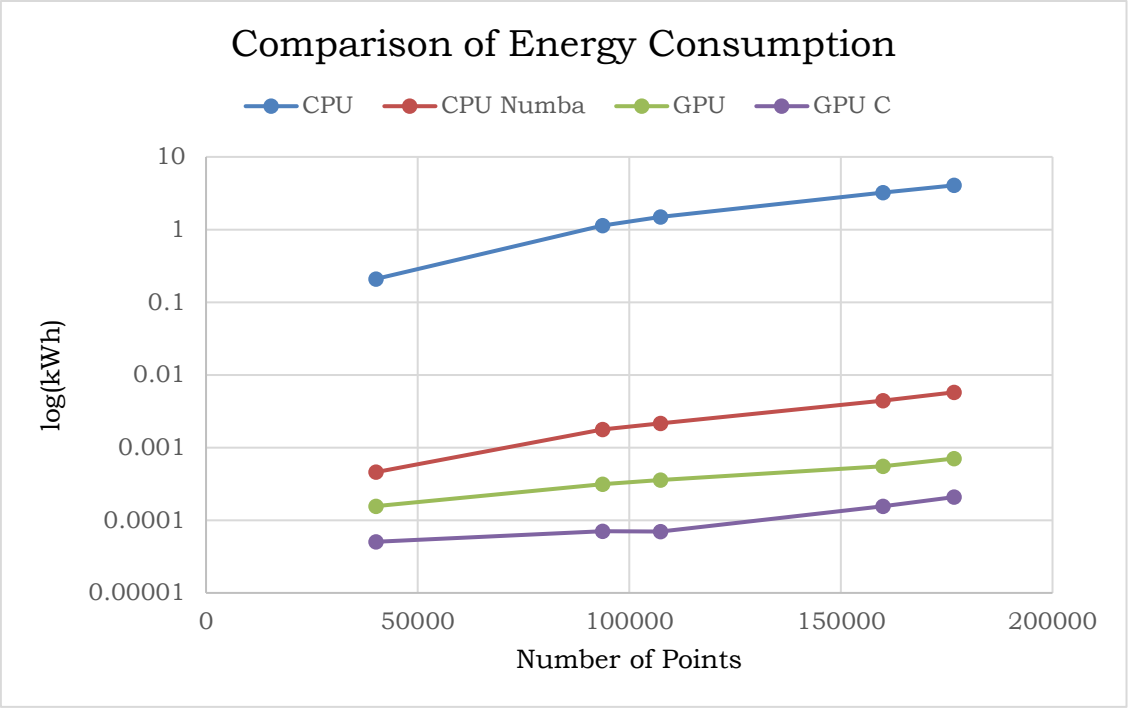


Figura 45: Comparación del consumo energético estimado para distintas versiones del algoritmo.

Como se observa, las versiones paralelas en GPU presentan un consumo energético inferior, lo que contribuye a una ejecución más sostenible. Esta eficiencia energética se alinea con los principios del ODS 12 (Producción y consumo responsables).

8.5 Relación con los Objetivos de Desarrollo Sostenible

El proyecto se relaciona con varios Objetivos de Desarrollo Sostenible de la Agenda 2030:

- ODS 9 (Industria, innovación e infraestructura): el trabajo contribuye al desarrollo de tecnologías avanzadas aplicables a sectores industriales y científicos.
- ODS 12 (Producción y consumo responsables): la optimización favorece un uso más eficiente de los recursos energéticos.
- ODS 3 (Salud y bienestar): la aceleración del análisis de imágenes puede apoyar investigaciones biomédicas y procesos de diagnóstico.
- ODS 13 (Acción por el clima): el código desarrollado puede aplicarse al análisis de datos ambientales y climáticos.

Siguiendo esos objetivos, durante el desarrollo del proyecto se han tomado decisiones dirigidas a aumentar el impacto positivo. Entre ellas destacan la priorización del rendimiento para reducir el tiempo de ejecución y, por tanto, el consumo energético; el uso de herramientas y lenguajes accesibles (Python y C); y la validación del algoritmo con datos reales.

9 Bibliografía

[1] Böhm, Christian and Noll, Robert and Plant, Claudia and Wackersreuther, Bianca. (2009). *Density-based clustering using graphics processors*. 661-670. 10.1145/1645953.1646038.

[2] Andrade, Guilherme and Ramos, Gabriel and Madeira, Daniel and Oliveira, Rafael and Ferreira, Renato and Rocha, Leonardo. *G-DBSCAN: A GPU accelerated algorithm for density-based clustering. Procedia Computer Science*. 18. 369-378. 10.1016/j.procs.2013.05.200.

[3] Ester, M. and H. peter Kriegel and J. S and X. Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise, AAAI Press*, 1996, pp. 226–231.

[4] Saeedi Emadi, H. and Mazinani, S. M. *A Novel Anomaly Detection Algorithm Using DBSCAN and SVM in Wireless Sensor Networks*, Wireless Personal Communications, vol. 98, pp. 2025-2035, 2018.

[5] Song, C. and Cui, J. and Cui, Y., et al., *Integrated STL-DBSCAN Algorithm for Online Hydrological and Water Quality Monitoring Data Cleaning*, Environmental Modelling & Software, vol. 183, 106262, 2024.

[6] Y. Yang, B. Lian, L. Li, C. Chen and P. Li, "DBSCAN Clustering Algorithm Applied to Identify Suspicious Financial Transactions," 2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Shanghai, China, 2014, pp. 60-65, doi: 10.1109/CyberC.2014.89

[7] Md Rezaul Karim, Oya Beyan, Achille Zappa, Ivan G Costa, Dietrich Rebholz-Schuhmann, Michael Cochez, Stefan Decker, Deep learning-based clustering approaches for bioinformatics, *Briefings in Bioinformatics*, Volume 22, Issue 1, January 2021, Pages 393–415, https://doi.org/10.1093/bib/bbz170

[8] Pacheco, Peter. 2011. *An Introduction to Parallel Programming* (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[9] NVIDIA, *CUDA C++ Programming Guide*, versión 13.0, NVIDIA Corporation, 2025. Disponible: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[10] Gruy, Frédéric. *Inertia tensor as morphological descriptor for aggregation dynamics. Colloids and Surfaces A: Physicochemical and Engineering Aspects*. 482. 10.1016/j.colsurfa.2015.04.034.

[11] "*Radius of gyration*" in IUPAC Compendium of Chemical Terminology, 3rd ed. (International Union of Pure and Applied Chemistry, 2006), Online version 3.0.1, 2019. https://doi.org/10.1351/goldbook.R05121

[12] NSF Unidata, netCDF, version 4.9, [software]. Boulder, CO, USA: UCAR / NSF Unidata Program Center, 2023. doi:10.5065/D6H70CW6.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*, Third Edition (3rd. ed.). The MIT Press.

[14] LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, Comp. Phys. Comm., 271, 108171 (2022) https://doi.org/10.1016/j.cpc.2021.108171

[15] Humphrey, W., Dalke, A. and Schulten, K., "VMD - Visual Molecular Dynamics", J. Molec. Graphics, 1996, vol. 14, pp. 33-38. http://www.ks.uiuc.edu/Research/vmd/

[16] M. Pachitariu, M. Rariden, and C. Stringer, “Cellpose-SAM: superhuman generalization for cellular segmentation,” bioRxiv, 2025, doi: https://doi.org/10.1101/2025.04.28.651001

[17] Allen, M. P.; D. J. Tildesley (1987). Computer Simulation of Liquids. Oxford: Clarendon Press.

10 Anexos

10.1 Repositorio del proyecto

A continuación, se detalla el contenido del repositorio del proyecto. https://github.com/rlombamoreno/TFG_DBSCAN

En el encontramos:

src/ : Directorio con los ficheros de código fuente del programa.

src/cpu_dbscan.py : Implementación secuencial del algoritmo DBSCAN en CPU

src/cpu_dbscan_numba.py: Versión optimizada en CPU mediante Numba.

src/gpu_dbscan.py: Implementación paralela en GPU del algoritmo DBSCAN en Python.

src/gpu_dbscan_ctypes.py: Implementación paralela en GPU del algoritmo DBSCAN en Python usando la versión optimizada en C mediante ctypes en la fase de propagación del agregado.

src/dbscan.cu: Código CUDA que implementa la propagación de agregados en GPU.

src/libdbscan.so: Biblioteca compilada en C/CUDA utilizada por la versión GPU optimizada.

pictures/ : Directorio de las fotos y archivos de datos usados en las pruebas

results/ : Directorio de los resultados de las simulaciones realizadas en las pruebas

Makefile: Fichero utilizado por make para compilar el archivo src/dbscan.cu

README.md : Fichero con los requisitos y las instrucciones de descarga e instalación de la aplicación.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jan 14 12:49:07 CET 2026
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)