ugr | Universidad de Granada

TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

DIFFERENTIAL BOX-COUNTING 4D EN GPU

Implementación y estudio experimental

Autor

Javier Martín Alcalde

Director

Juan Ruiz de Miras

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2024

Differential Box-counting 4D EN GPU: Implementación y estudio experimental

Javier Martín Alcalde

Palabras clave: Dimensión fractal, Differential Box-Counting, CUDA, GPU, procesamiento paralelo, implementación secuencial, análisis de datos.

Resumen

En este Trabajo de Fin de Grado se ha implementado y estudiado el algoritmo Differential Box-Counting para el cálculo de la dimensión fractal en matrices de cuatro dimensiones, con el objetivo de mejorar la precisión y eficiencia en el análisis de datos complejos. Se ha desarrollado una versión secuencial en C++ y una versión paralela utilizando la tecnología CUDA para aprovechar la capacidad de procesamiento masivo de las GPUs. El trabajo incluye un análisis comparativo de los tiempos de ejecución entre ambas implementaciones y la aplicación del algoritmo a datos reales de electroencefalografía, evaluando su efectividad en la clasificación de estados de conciencia.

Differential Box-counting 4D IN GPU: Implementation and Experimental Study

Javier Martín Alcalde

Keywords: Fractal dimension, Differential Box-Counting, CUDA, GPU, parallel processing, sequential implementation, data analysis.

Abstract

In this Final Degree Project, the Differential Box-Counting algorithm has been implemented and studied for calculating the fractal dimension in four-dimensional matrices, with the objective of improving accuracy and efficiency in the analysis of complex data. A sequential version in C++ and a parallel version using CUDA technology have been developed to leverage the massive processing capabilities of GPUs. The work includes a comparative analysis of execution times between both implementations and the application of the algorithm to real electroencephalography data, evaluating its effectiveness in classifying states of consciousness.

Yo, **Javier Martín Alcalde**, alumno de la titulación Ineniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de telecomunicación de la Universidad de Granada**, con DNI 20995539N, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Javier Martín Alcalde

Granada a 24 de junio de 2024.

*

D. **Juan Ruiz de Miras**, Profesor del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informo:

Que el presente trabajo, titulado ***Differential Box-counting 4D EN GPU: Implementación y estudio experimental***, ha sido realizado bajo su supervisión por **Javier Martín Alcalde**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

El director:

```
Firmado por RUIZ DE MIRAS JUAN -
***9504** el día 24/06/2024 con un
certificado emitido por AC FNMT
Usuarios
```

Juan Ruiz de Miras

Índice de Contenidos

Índice de Listados

Índice de figuras

Índice de cuadros

1. Introducción

El cálculo de la dimensión fractal [1] es una herramienta fundamental utilizada en el campo de la visión por computador y el análisis de imágenes debido a su capacidad de caracterizar la rugosidad y la auto-similitud de los objetos dentro de una imagen. La dimensión fractal permite describir objetos irregulares y complejos que no pueden ser representados adecuadamente mediante la geometría euclidiana tradicional. Introducida por Mandelbrot en 1983 [2], la geometría fractal ha permitido avances significativos en diversas aplicaciones, tales como la segmentación de texturas, el reconocimiento de formas y el análisis de redes complejas.

Para mejorar la precisión y eficiencia de los métodos de análisis de imágenes, es esencial estimar de manera precisa la dimensión fractal. El método Differential Box-Counting (DBC) [3] destaca entre las diversas técnicas por su simplicidad y efectividad. El algoritmo DBC estima la dimensión fractal interpretando una imagen en escala de grises y contando el número de cajas necesarias para cubrir la superficie en diferentes escalas.Este algoritmo se explicará detalladamente en la Sección 3.

El algoritmo DBC ha sido ampliamente utilizado para el análisis de imágenes en dos dimensiones. En V. Lakshmi Praba1, S. Esakkiammal [4] podemos observar un estudio sobre la textura de las imágenes. Si ampliamos a tres dimensiones, encontramos también análisis de texturas [5], segmentación de imágenes [6], procesamiento de activaciones cerebrales [7], entre otros. Este trabajo se centra en la implementación del algoritmo DBC para el cálculo de la dimensión fractal en matrices de cuatro dimensiones, lo que permitirá utilizar dicho algoritmo en aplicaciones que requieran el análisis de imágenes 3D que evolucionan en el tiempo, un desafío que requiere abordar la complejidad adicional de los datos multidimensionales.

1.1. Objetivos

El objetivo de este trabajo fin de grado es proporcionar una solución eficiente y escalable para calcular la dimensión fractal de modelos representados por matrices de cuatro dimensiones. Primero se realizará una implementación secuencial en C++, lo que nos va a servir como base para posteriormente realizar una implementación del algoritmo utilizando la tecnología CUDA [8], con el fin de aprovechar el paralelismo que proporcionan las GPUs actuales así como la arquitectura SIMD (Single Instruction Multiple Data) presente en las tarjetas gráficas.

Se parte de un algoritmo inicial implementado por el tutor del proyecto, el profesor D. Juan Ruiz de Miras [7] para el procesamiento de matrices en tres dimensiones. El objetivo del proyecto es ampliar el algoritmo para procesar matrices en cuatro dimensiones, tanto en la versión secuencial usando C++ como en la paralelización con CUDA. Se espera proporcionar una herramienta robusta y eficiente para el análisis

de datos complejos en diversas aplicaciones.

El trabajo consistirá en proporcionar una solución eficiente y escalable para calcular la dimensión fractal de matrices de 4 dimensiones. Primero se extenderá a 4D el algoritmo ya existente para procesar nubes de puntos en 3D de forma secuencial utilizando C++. Esto nos va a servir como base para posteriormente paralelizar el algoritmo utilizando CUDA para aprovechar la capacidad de procesamiento paralelo de las tarjetas gráficas. A continuación se tomarán tiempos de ejecución con diferentes ejemplos de prueba para realizar un estudio sobre la eficiencia de la implementación usando la GPU. Finalmente, se aplicará el algoritmo en casos reales de encefalografía, analizando nubes de puntos que representan activaciones cerebrales en pacientes, y se realizará un estudio comparativo para clasificar estados de conciencia identificando diferencias significativas.

1.2. Metodología

Este trabajo experimental sigue una metodología estructurada para alcanzar los objetivos propuestos. A continuación, se detallan las principales etapas del proceso:

1. **Estudio del Algoritmo Differential Box-Counting (DBC):** En primer lugar, se realizará un estudio exhaustivo del algoritmo DBC para comprender su funcionamiento y los fundamentos matemáticos que lo sustentan. También se analizará el código del algoritmo inicial implementado por el profesor D. Juan Ruiz de Miras para el procesamiento de matrices en tres dimensiones.

2. **Implementación Secuencial en C++:** Se desarrollará una implementación secuencial del algoritmo DBC para el cálculo de la dimensión fractal en matrices de cuatro dimensiones utilizando el lenguaje de programación C++. Esta versión secuencial sirve como base y punto de comparación para la posterior implementación paralela. Durante esta etapa, se realizarán pruebas y ajustes necesarios para asegurar la correcta funcionalidad del algoritmo.

3. **Estudio de la Tecnología CUDA:** Una ver finalizada la etapa de la implementación secuencial se llevará a cabo un estudio detallado de la tecnología CUDA. Este estudio incluye la revisión de la arquitectura de CUDA, el modelo de programación, las extensiones de sintaxis y las funciones y librerías disponibles. El objetivo es familiarizarse con las herramientas y técnicas necesarias para paralelizar el algoritmo DBC de manera eficiente.

4. **Implementación Paralela en CUDA:** Con el conocimiento adquirido sobre CUDA, se procederá a implementar el algoritmo DBC utilizando esta tecnología. Se deberá optimizar el código para aprovechar la capacidad de procesa-

miento masivo y la arquitectura de las GPUs. Al igual que con la implementación secuencial, se realizarán pruebas para verificar la correcta ejecución y se optimizará el rendimiento del algoritmo.

5. **Comparativa de Tiempos de Ejecución:** Una vez desarrolladas las versiones secuencial y paralela del algoritmo, se realizarán pruebas comparativas de los tiempos de ejecución utilizando diferentes ejemplos. Se medirá el rendimiento de ambas implementaciones y se analizará la eficiencia y escalabilidad de la versión paralela en comparación con la versión secuencial.

6. **Aplicación en Casos Reales:** Finalmente, se aplicará el algoritmo paralelizado en CUDA a casos reales. En particular, se analizarán nubes de puntos que representan activaciones cerebrales con datos de encefalografía sobre sujetos dormidos y despiertos. Se realizará un estudio comparativo de los datos obtenidos para identificar posibles diferencias y patrones relevantes entre los diferentes grupos estudiados.

2. Plan de proyecto

Este capítulo describe el plan de proyecto. Se detallan las tareas a realizar junto con el tiempo estimado para cada una y un diagrama de Gantt que ilustra la planificación temporal del proyecto.

En el grado de Ingeniería Informática, el trabajo de fin de grado tiene una carga de 12 créditos ECTS [9]. Cada crédito ECTS equivale a 25 horas de trabajo del alumno, por lo que obtenemos que el proyecto debe ser completado en 300 horas.

2.1. Descripción de tareas

2.1.1. Planificación del proyecto

Se lleva a cabo la planificación detallada del proyecto, incluyendo la definición de tareas, asignación de tiempos y recursos necesarios. Se realiza un diagrama de Gantt que ilustre la planificación.

Tiempo Estimado: 10 horas.

2.1.2. Estudio del algoritmo Differential Box-counting

Se realiza un estudio exhaustivo del algoritmo DBC, revisando la literatura existente y comprendiendo los fundamentos matemáticos y computacionales del algoritmo.

Tiempo Estimado: 30 horas.

2.1.3. Implementación secuencial en C++

Desarrollo de una implementación secuencial del algoritmo DBC para el cálculo de la dimensión fractal en matrices de cuatro dimensiones utilizando el lenguaje de programación C++.

Tiempo Estimado: 40 horas.

2.1.4. Estudio de la tecnología CUDA

Realización de un estudio detallado sobre la tecnología CUDA, incluyendo la arquitectura, el modelo de programación y las herramientas disponibles.

Tiempo Estimado: 30 horas.

2.1.5. Implementación paralela usando CUDA

Desarrollo de una implementación paralela del algoritmo DBC utilizando CUDA para aprovechar la capacidad de procesamiento masivo de las GPUs.

Tiempo Estimado: 90 horas.

2.1.6. Estudio comparativo de tiempos de ejecución entre CPU y GPU

Análisis comparativo de los tiempos de ejecución entre la implementación secuencial y la paralela para evaluar la eficiencia y escalabilidad de cada versión. También se ejecuta la versión paralela del algoritmo en un servidor.

Tiempo Estimado: 20 horas.

2.1.7. Aplicación en casos reales

Aplicación del algoritmo paralelizado en CUDA a datos reales para identificar posibles diferencias y patrones relevantes. Se estudia el poder de clasificación de estados de conciencia a partir de datos de encefalografía.

Tiempo Estimado: 20 horas.

2.1.8. Redacción de la memoria del proyecto

Redacción continua de la memoria del proyecto, documentando cada etapa del desarrollo y los resultados obtenidos.

Tiempo Estimado: 40 horas (se distribuyen a lo largo de todo el proyecto).

2.1.9. Preparación de la defensa

Preparación de la presentación y defensa del proyecto ante el tribunal. Incluye la creación de diapositivas, práctica de la presentación y revisión de posibles preguntas.

Tiempo estimado: 20 horas.

2.2. Diagrama de Gant

A continuación se muestra el diagrama de Gantt que ilustra la planificación temporal del proyecto, distribuyendo las tareas a lo largo de las 300 horas disponibles:

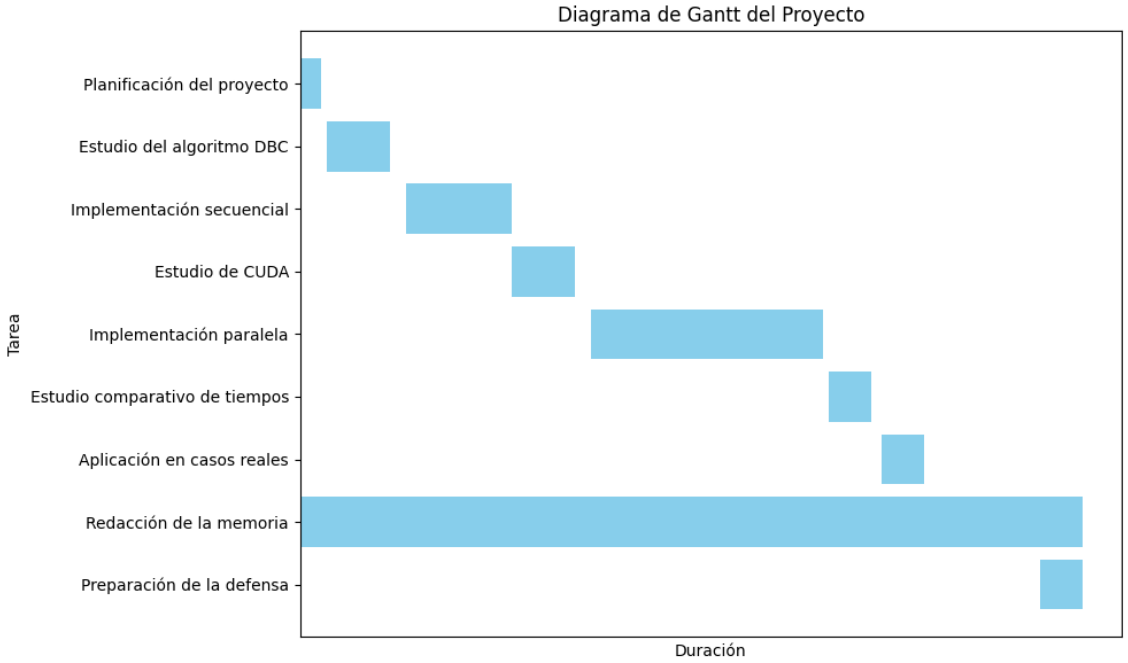

Figura 1: Diagrama de gantt que ilustra la planificación del proyecto.

2.3. Recursos utilizados

En este subapartado se describen los recursos de hardware empleados para la implementación y ejecución del algoritmo DBC en sus versiones secuencial y paralela. Los recursos se dividen en dos categorías principales: los utilizados en el portátil personal del estudiante, Javier Martín, y los utilizados en el servidor de la E.T.S.I.I.T.

2.3.1. Portátil personal

Para el desarrollo y ejecución de la versión secuencial del algoritmo en C++, he utilizado mi portátil, un Asus Rog Strix que cuenta con las siguientes características:

- Procesador: AMD Ryzen 7 4800H
  - Velocidad: 2.9 GHz
  - Número de núcleos: 8
- Memoria RAM: 16GB
- Sistema Operativo: Windows 11

Además, para la ejecución de la versión paralela del algoritmo utilizando CUDA, mi portátil está equipado con una tarjeta gráfica NVIDIA:

- GeForce RTX 3060

- Núcleos NVIDIA CUDA: 3584.
- Memoria: 12 GB (GDDR6)

2.3.2. Servidor

Como se ha comentado en la planificación del proyecto, también se va a ejecutar la implementación paralela del algoritmo en el servidor de supercomputación Isicomputing.ugr.es, perteneciente al Departamento de Lenguajes y Sistemas Informáticos de la UGR. Este servidor nos permite realizar una comparación de los tiempos de ejecución más amplia al contar con un entorno hardware más potente. Las características del servidor son las siguientes:

- Sistema operativo: Debian Linux 5.10
- CPU: Intel Xeon Silver 4210 2.20GHz, 96GB de RAM
- GPU: NVIDIA Tesla V100 PCIexpress, 5120 CUDA cores, 32GB de memoria global

3. El algoritmo

El algoritmo Differential Box-Counting (DBC) [3] es una metodología utilizada para calcular la dimensión fractal (FD) [1] de una imagen en escala de grises. La FD es una medida cuantitativa que describe cómo una estructura fractal llena el espacio a diferentes escalas. Esta característica es muy valiosa porque proporciona una medida cuantitativa de la complejidad y rugosidad de objetos y fenómenos que no pueden ser descritos adecuadamente por la geometría auclidiana tradicional, lo que nos proporciona una comprensión más profunda y precisa de los datos utilizados, mejorando la capacidad de análisis, modelado y toma de decisiones. Este método fue introducido por Sarkar y Chaudhuri [10] y se basa en el concepto de contar el número de cajas necesarias para cubrir la superficie de la imagen a diferentes resoluciones.

El algoritmo DBC se aplica a imágenes en escala de grises, las cuales se consideran como superficies tridimensionales donde las coordenadas x e y representan la posición del píxel y la coordenada z representa el nivel de gris (intensidad) del píxel. A continuación, se detallan los pasos del algoritmo para el caso de imágenes 2D:

1. División de la imagen: La imagen de tamaño $M \times M$ píxeles se divide en una cuadrícula de tamaño $s \times s$ píxeles, donde s es un tamaño de caja que varía entre 2 y $M/2$.

2. Consideración de la imagen como superficie 3D: La imagen en escala de grises se trata como una superficie tridimensional. El tercer eje (eje z) está representado por los niveles de gris de los píxeles de la imagen, siendo G el número total de niveles de gris.

3. Definición de las cajas: Para cubrir la superficie de la imagen sobre cada celda $s \times s$, se define un conjunto de cajas de tamaño $s \times s \times h$, donde h es tal que $[G/h] = [M/s]$. Esto asegura que la dimensión de las cajas en el eje z se ajuste correctamente a los niveles de gris de la imagen.

4. Cálculo de $n_r(i, j)$: Para cada celda (i, j) de la cuadrícula, se calcula $n_r(i, j)$ utilizando la fórmula:

$$n_r(i, j) = k - l + 1 \tag{1}$$

   donde k y l son los números de las cajas en las que se encuentran los niveles de gris máximo y mínimo dentro de la celda, respectivamente. En la Figura 2, se muestra un ejemplo de una cuadrícula 2×2 con $k = 3$ y $l = 1$, resultando en $n_r(i, j) = 3 - 1 + 1 = 3$.

5. Cálculo de N_r: Se calcula N_r, que es el número total de cajas necesarias para cubrir la superficie de la imagen, sumando $n_r(i, j)$ para todas las celdas (i, j)

de la imagen:

$$N_r = \sum_{i,j} n_r(i,j) \tag{2}$$

donde r varía desde $\frac{2}{M}$ hasta $\frac{1}{2}$.

6. Estimación de la dimensión fractal: Finalmente, la dimensión fractal de la imagen se obtiene mediante la regresión lineal de $\log(N_r)$ frente a $\log(1/r)$ para varios valores de r. La pendiente de la línea ajustada representa la dimensión fractal (FD) de la imagen.

$$\text{FD} = \frac{\log(N_r)}{\log(1/r)} \tag{3}$$

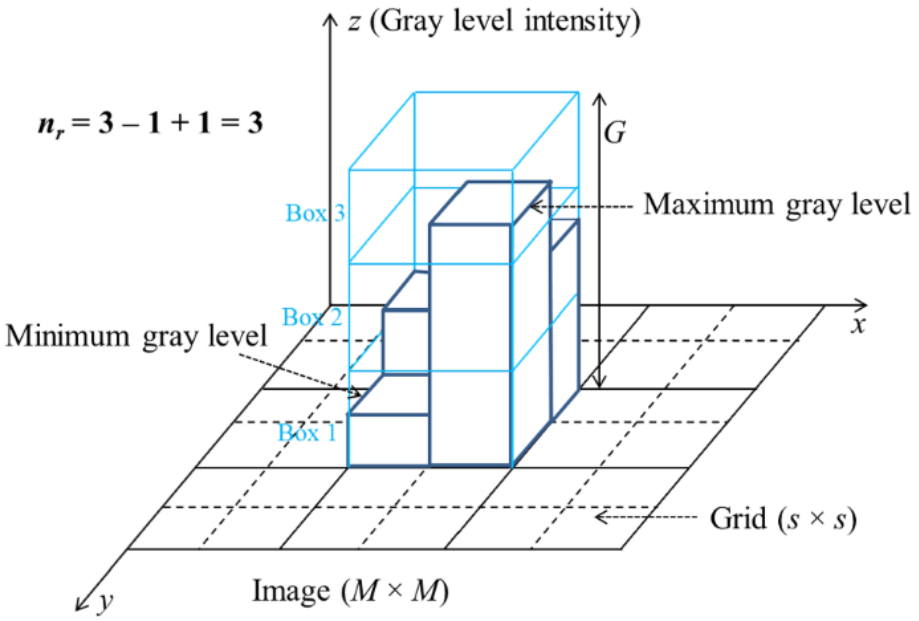

Figura 2: Determinación de N_r mediante el algoritmo DBC. Cálculo detallado de n_r para una cuadrícula de 2×2. [7]

3.1. Características del algoritmo

El algoritmo sobre el que se sustenta este proyecto tiene 3 características destacables:

- Método de escaneo: Se trata de la manera que tenémos de recorrer la matriz, en el algoritmo utilizado para este proyecto se utiliza un método de escaneo de cuadrícula fija ("fixed grid scans"*en inglés*) . Esto quiere decir que a la hora de subdividir la estructura de datos en cajas, se hará de manera que para cada tamaño de caja seleccionado, un elemento de la matriz pertenecerá únicamente a una caja, sin que exista la posibilidad de que un elemento este en dos cajas para una misma subdivisión.

- Evaluación de Cajas: Una vez que tenemos la estructura de datos dividida en cajas del mismo tamaño, necesitamos establecer una regla para evaluar cada caja. En el algoritmo DBC, esto se realiza calculando los valores máximos y mínimos dentro de cada caja y utilizando la diferencia entre estos valores para determinar el número de cajas.

- Aumento del tamaño de la caja: En cada iteración el tamaño de la caja aumenta siguiendo las potencias de 2 (2,4,8...) hasta llegar a la mitad del tamaño de las dimensiones de la matriz. Se trabaja con matrices cuadradas con tamaños de fila o columna potencia de 2.

3.2. Pseudocódigo

Como se ha comentado anteriormente, el cálculo de r no supone ninguna complicación, ya que conociendo el tamaño de la matriz cuadrada, simplemente consiste en crear una lista cuyos valores sean las potencias de 2 hasta la mitad del tamaño de la matriz.

Por otra parte, sí hay que tener especial cuidado con el cálculo de los valores máximos y mínimos de cada caja, para determinar el número de las mismas.

Como se puede observar en el listado 1, el algoritmo toma como entrada una imagen I de tamaño $M \times M$, el número de niveles de gris G, y un array Nr donde se almacenarán los resultados para cada tamaño de caja s. Se comienza inicializando los buffers $Imax$ e $Imin$ con los niveles de gris de la imagen original I y se entra en el bucle principal.

Una vez dentro del bucle, el tamaño de la cuadrícula s comienza en 2 y se duplica en cada iteración hasta alcanzar $M/2$. Para cada iteración el algoritmo calcula los valores de gris máximos y mínimos para cada celda $s \times s$ y se almacenan en $Imax$ e $Imin$. Se calcula n_r para cada celda (i, j) y se almacena en el array Nr. Por último se actualiza el tamaño de la cuadrícula s.

```
1  DBC(const unsigned char* I, const int M, const int G, unsigned long
       * Nr) {
2  Imax[M][M] = Imin[M][M] = I
3  s = 2;
4  size = M;
5  Nri = 0;
6  while (size > 2) {
7      h = (G * s) / M;
8
9      for (unsigned long i = 0; i < (M - 1); i += s) {
10         for (unsigned long j = 0; j < (M - 1); j += s) {
11             Imax[i][j] = max(Imax[i][j], Imax[i,j+s/2][j], Imax[i+s
                   /2][j], Imax[i+s/2][j+s/2]);
```

```
12              Imin[i][j] = min(Imin[i][j], Imin[i,j+s/2][j], Imin[i+s
                    /2][j], Imin[i+s/2][j+s/2]);
13
14              Nr[Nri] = Nr[Nri] + (Imax[i][j]/h - Imin[i][j]/h + 1);
15          }
16      }
17      Nri++;
18      s = s * 2;
19      size = s / 2;
20  }
```

Listing 1: Pseudocódigo del algoritmo Differential Boox-Counting para matrices bidimensionales.

3.3. Ejemplo para matriz bidimensional

Con el objetivo de clarificar el funcionamiento del algoritmo se muestra un ejemplo de ejecución del algoritmo para una matriz bidimensional 16×16 representada en la figura 3. En este caso de ejemplo se toman valores de gris entre 0 y 9. En este ejemplo sólo se muestra el cálculo de $Imax$, siendo el cálculo de $Imin$ completamente análogo.

1. En la imagen (a) se muestra la matriz original de 16×16 píxeles.

2. Cálculo para $s = 2$: Se divide la imagen en cuadrículas de 2×2 píxeles. Para cada cuadrícula se determina el nivel de gris máximo y mínimo y se almacena en la posición correspondiente de la misma matriz. El resultado de este paso se muestra en la imagen (b), donde los valores rojos indican las posiciones $(0, 0)$ de cada cuadrícula de 2×2, que almacenan el máximo de cada caja de tamaño 2.

3. Cálculo para $s = 4$: Se divide la imagen en cuadrículas de 4×4 píxeles. Para cada cuadrícula, se determina el nivel de gris máximo utilizando los valores previamente calculados para $s = 2$. En este caso, se compara el nivel de gris máximo de las cuatro sub-cuadrículas de 2×2 que forman la cuadrícula de 4×4 El resultado de este paso se muestra en la subfigura (c), donde los valores rojos indican las posiciones (0, 0) de cada cuadrícula de 4×4, que alacenan el máximo de cada caja de tamaño 4.

4. Cálculo para $s = 8$: Se divide la imagen en cuadrículas de 8×8 píxeles. Para cada cuadrícula, se determina el nivel de gris máximo utilizando los valores previamente calculados para $s = 4$. En este caso, se compara el nivel de gris máximo de las cuatro sub-cuadrículas de 4×4 que forman la cuadrícula de

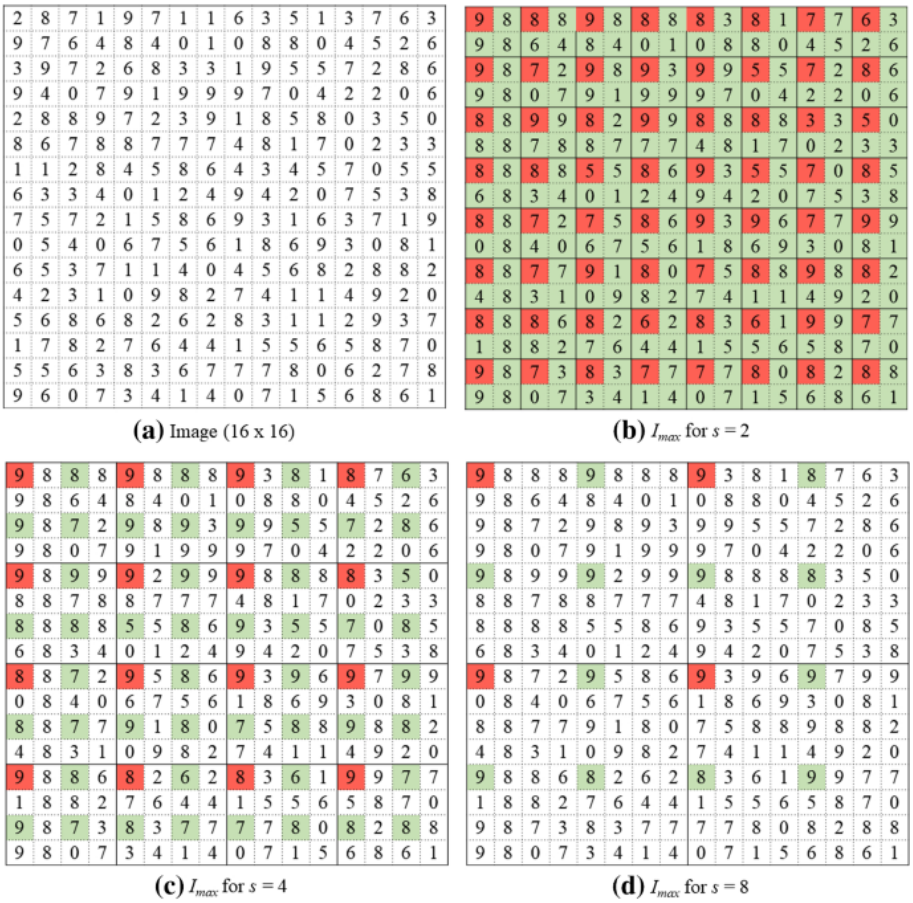

Figura 3: Ejemplo del cálculo de niveles máximos de gris para una matriz bidimensional. Imagen tomada de [7].

8×8. El resultado de este paso se muestra en la subfigura (d), donde los valores rojos indican las posiciones (0, 0) de cada cuadrícula de 8×8, que almacenan el máximo de cada caja de tamaño 8.

3.4. Orden de complejidad del algoritmo

El análisis de la complejidad [11] del algoritmo Differential Box-Counting (DBC) para matrices de cuatro dimensiones es crucial para entender el rendimiento y los recursos computacionales necesarios para su ejecución. En este apartado, se desglosa detalladamente el orden de complejidad del algoritmo en el contexto de una matriz 4D.

En el caso de una matriz de cuatro dimensiones, consideramos una imagen con dimensiones $M \times M \times M \times M$, donde cada punto de la matriz tiene un valor de intensidad que puede variar entre 0 y $G - 1$. El algoritmo DBC se aplica a esta matriz para calcular la dimensión fractal, tratándola como una superficie en un espacio de cinco dimensiones (las cuatro dimensiones espaciales más la dimensión de intensidad).

El algoritmo comienza dividiendo la matriz original en submatrices de tamaño $s \times s \times s \times s$, donde s es el tamaño de la submatriz que varía desde 2 hasta $M/2$. Esta división resulta en la generación de $(\frac{M}{s})^4$ submatrices.

A continuación, para cada submatriz, se determinan los valores máximos y mínimos de intensidad, lo que implica recorrer completamente cada submatriz. Esto requiere s^4 operaciones para cada submatriz.

Se continúa calculando el número de cajas necesarias para cubir la diferencia entre los niveles de intensidad máximo y mínimo, se calcula N_r y se estima la dimensión fractal mediante la regresión lineal de $log_2(M)$ frente a $log(\frac{1}{r})$ para varios valores de r.

Para calular la complejidad también debemos tener en cuenta el número de niveles de s, el cual varía de 2 hasta $M/2$. Si consideramos todos los tamaños posibles de s en potencias de 2 hay $log_2(M)$.

Combinando estos factores, el orden de complejidad del algoritmo DBC para una matriz 4D se puede estimar multiplicando el número de submatrices por el costo de calcular los niveles de intensidad máximo y mínimo en cada submatriz, y luego por el número de niveles de s:

$$O(\mathrm{DBC}_{4D}) = \sum_{s=2}^{M/2} \left(\frac{M}{s}\right)^4 \cdot s^4 = \sum_{s=2}^{M/2} M^4 = M^4 \cdot \log_2(M) \tag{4}$$

Por lo tanto, la complejidad del algoritmo DBC para matrices de cuatro dimensiones es:

$$O(\mathrm{DBC}_{4D}) = M^4 \cdot \log_2(M) \tag{5}$$

El orden de complejidad obtenido indica que el algoritmo es altamente dependiente del tamaño de la matriz. A medida que M crece, el número de operaciones necesarias aumenta de manera exponencial. Esta complejidad hace que la implementación secuencial del algoritmo en C++ pueda volverse impráctica para matrices de gran tamaño debido a los altos requerimientos computacionales. Esta limitación subraya la importancia de utilizar técnicas de paralelización para manejar eficientemente grandes volúmenes de datos y mejorar el rendimiento computacional.

4. Implementación secuencial

En este apartado se describe el proceso de implementación secuencial del algoritmo Differential Box-Counting (DBC) para el procesamiento de matrices en cuatro dimensiones. Para llevar a cabo esta implementación, se ha elegido el lenguaje de programación C++ [12] debido a su eficiencia y control sobre los recursos de hardware, lo cual es crucial para manejar grandes volúmenes de datos y operaciones intensivas en cómputo.

El punto de partida para esta implementación es el trabajo previo realizado por el profesor D. Juan Ruiz de Miras [7], quien desarrolló un algoritmo para el procesamiento de matrices en tres dimensiones. Este algoritmo inicial, escrito en C++, se ha extendido y adaptado para procesar matrices en cuatro dimensiones, lo que implica no solo un aumento en la complejidad del procesamiento, sino también la necesidad de optimizar el código para mantener la eficiencia.

En esta sección se detallará cómo se ha llevado a cabo esta ampliación, incluyendo la estructura del código, las modificaciones necesarias para manejar datos en cuatro dimensiones y las técnicas utilizadas para asegurar una implementación robusta y eficiente. Se proporcionarán fragmentos de código relevantes y se explicarán los conceptos clave para entender cómo se ha logrado la extensión del algoritmo a cuatro dimensiones.

4.1. Estructura del código

Como se puede observar en el listado 2, el código de la implementación secuencial se estructura en varias partes clave:

1. Inicialización de Buffers: Se inicializan dos buffers ($Imax$ e $Imin$) que almacenarán los valores máximos y mínimos de intensidad para cada tamaño de submatriz s.

2. Bucle Principal de Cálculo: Se utiliza un bucle while que itera sobre diferentes tamaños de submatriz s, comenzando en 2 y duplicándose en cada iteración hasta alcanzar $M/2$.

3. Cálculo de Máximos y Mínimos: Dentro del bucle principal, se calculan los valores máximos y mínimos de intensidad para cada submatriz de $s \times s \times s \times s$, utilizando comparaciones entre los valores de intensidad.

4. Actualización de Resultados: Los resultados de los cálculos se almacenan en el array Nr, que contiene el número de cajas necesarias para cubrir la superficie de la matriz para cada tamaño de submatriz s.

5. Liberación de Memoria: Finalmente, se liberan los recursos de memoria asignados a los buffers $Imax$) e $Imin$.

```
void DBC_4D_CPU(const unsigned char* I, const int M, const int G,
    unsigned long* Nr) {

    unsigned char* Imax = new unsigned char[M * M * M * M];
    unsigned char* Imin = new unsigned char[M * M * M * M];
    unsigned char maxv, minv;
    memcpy(Imax, I, sizeof(unsigned char) * M * M * M * M);
    memcpy(Imin, I, sizeof(unsigned char) * M * M * M * M);
    unsigned int s = 2, size = M;
    unsigned char Nri = 0;

    while (size > 2) {
        int sp = ceilf(((float)(G << (Nri + 1))) / (float)M);
        float invsp = 1.0 / sp;
        int sm = s >> 1;
        unsigned long sum = 0;
        unsigned long iM, kMM, lMMM, ismM, ksmMM, lsmMMM
        Nr[Nri] = 0;
        for (unsigned long l = 0; l < (M - 1); l += s) {
            lMMM = l * M * M * M;
            lsmMMM = (l + sm) * M * M * M;
            for (unsigned long k = 0; k < (M - 1); k += s) {
                kMM = k * M * M;
                ksmMM = (k + sm) * M * M;
                for (unsigned long i = 0; i < (M - 1); i += s) {
                    iM = i * M;
                    ismM = (i + sm) * M;
                    for (unsigned long j = 0; j < (M - 1); j += s){
                        max_min_if(Imax[lMMM + kMM + iM + j],
                            Imax[lMMM + kMM + iM + (j+sm)],
                            Imax[lMMM + kMM + ismM + j],
                            Imax[lMMM + kMM + ismM + (j+sm)],
                            Imax[lMMM + ksmMM + iM + j],
                            Imax[lMMM + ksmMM + iM + (j+sm)],
                            Imax[lMMM + ksmMM + ismM + j],
                            Imax[lMMM + ksmMM + ismM + (j+sm)],
                            Imax[lsmMMM + kMM + iM + j],
                            Imax[lsmMMM + kMM + iM + (j+sm)],
                            Imax[lsmMMM + kMM + ismM + j],
                            Imax[lsmMMM + kMM + ismM + (j+sm)],
                            Imax[lsmMMM + ksmMM + iM + j],
                            Imax[lsmMMM + ksmMM + iM + (j+sm)],
                            Imax[lsmMMM + ksmMM + ismM + j],
                            Imax[lsmMMM + ksmMM + ismM + (j+sm)],
```

```
44                                  Imin[lMMM + kMM + iM + j],
45                                  Imin[lMMM + kMM + iM + (j + sm)],
46                                  Imin[lMMM + kMM + ismM + j],
47                                  Imin[lMMM + kMM + ismM + (j + sm)],
48                                  Imin[lMMM + ksmMM + iM + j],
49                                  Imin[lMMM + ksmMM + iM + (j + sm)],
50                                  Imin[lMMM + ksmMM + ismM + j],
51                                  Imin[lMMM + ksmMM + ismM + (j + sm)],
52                                  Imin[lsmMMM + kMM + iM + j],
53                                  Imin[lsmMMM + kMM + iM + (j + sm)],
54                                  Imin[lsmMMM + kMM + ismM + j],
55                                  Imin[lsmMMM + kMM + ismM + (j + sm)],
56                                  Imin[lsmMMM + ksmMM + iM + j],
57                                  Imin[lsmMMM + ksmMM + iM + (j + sm)],
58                                  Imin[lsmMMM + ksmMM + ismM + j],
59                                  Imin[lsmMMM + ksmMM + ismM + (j + sm)],
60                                 maxv, minv);
61                          Imax[kMM + iM + j] = maxv;
62                          Imin[kMM + iM + j] = minv;
63                          sum += ceilf((float)maxv * invsp) - ceilf((
                                float)minv * invsp) + 1;
64                      }
65                  }
66              }
67          }
68          Nr[Nri] = sum;
69          s <<= 1; // s *= 2;
70          size >>= 1; // size /= 2;
71          Nri++;
72      }
73      delete Imax;
74      delete Imin;
75  }
```

Listing 2: Implementación del algoritmo Differential Boox-Counting para matrices de cuatro dimensiones.

4.2. Detalles de implementación

A continuación se enumeran algunos detalles que se han tenido en cuenta a la hora de realizar la implementación del código:

- **Cálculo de máximos y mínimos**. La función 'max_min' se utiliza para comparar los valores de intensidad dentro de cada submatriz de $s \times s \times s \times s$ y determinar los valores máximo y mínimo. Este paso es crítico para calcular correctamente el número de cajas necesarias para cubrir la superficie en cada

iteración. Como se puede observar en el listado 3, se utilizan sentencias ifs para realizar el cálculo de dichos valores. También se ha implementado una función equivalente que inicializa un array con los valores estudiados e utiliza las funciones *max_element*() y *min_element*() de C++ pero los tiempos obtenidos son significativamente mayores.

- **Actualización de Resultados:** Los resultados de los cálculos se almacenan en el array Nr. Este array se actualiza en cada iteración del bucle principal, asegurando que se registren los resultados para cada tamaño de submatriz s.

```
1  void max_min(const unsigned char a, const unsigned char b, ...,
       const unsigned char p, const unsigned char a2, const unsigned
       char b2, ..., const unsigned char p2, unsigned char& maxv,
       unsigned char& minv) {
2      if ((a >= b) && (a >= c) && (a >= d) && (a >= e) && (a >= f) &&
           (a >= g) && (a >= h) && (a >= i) && (a >= j) && (a >= k) &&
           (a >= l) && (a >= m) && (a >= n) && (a >= o) && (a >= p))
          maxv = a;
3      else if ((b >= a) && (b >= c) && (b >= d) && (b >= e) && (b >=
          f) && (b >= g) && (b >= h) && (b >= i) && (b >= j) && (b >=
          k) && (b >= l) && (b >= m) && (b >= n) && (b >= o) && (b >=
          p)) maxv = b;
4      // Comparaciones adicionales omitidas para brevedad
5      else maxv = p;
6
7      if ((a2 <= b2) && (a2 <= c2) && (a2 <= d2) && (a2 <= e2) && (a2
           <= f2) && (a2 <= g2) && (a2 <= h2) && (a2 <= i2) && (a2 <=
          j2) && (a2 <= k2) && (a2 <= l2) && (a2 <= m2) && (a2 <= n2)
          && (a2 <= o2) && (a2 <= p2)) minv = a2;
8      // Comparaciones adicionales omitidas para brevedad
9      else minv = p2;
10 }
```

Listing 3: Función auxiliar para calcular el valor máximo y mínimo de una submatriz $s \times s \times s \times s$.

5. Tecnología CUDA

CUDA [8] son las siglas de Compute Unified Device Architecture, y es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA. Fue introducida por primera vez en 2006 con el objetivo de permitir a los desarrolladores aprovechar la capacidad de procesamiento masivo de las unidades de procesamiento gráfico (GPU) para realizar cálculos de propósito general, más allá de los gráficos.

La arquitectura CUDA transforma las GPUs, tradicionalmente utilizadas exclusivamente para el renderizado de gráficos, en procesadores de propósito general que pueden abordar una amplia gama de tareas computacionales. Antes de CUDA, el uso de las GPUs para computación general (conocido como GPGPU) era complicado y requería habilidades específicas en programación de gráficos. CUDA simplifica este proceso al proporcionar un modelo de programación basado en extensiones del lenguaje C, facilitando el desarrollo y ejecución de aplicaciones paralelas en la GPU.

El propósito principal de CUDA es mejorar el rendimiento de las aplicaciones computacionales mediante la paralelización masiva. Las GPUs tienen cientos o incluso miles de núcleos de procesamiento que pueden ejecutar miles de hilos en paralelo, lo que permite manejar grandes volúmenes de datos y realizar cálculos intensivos de manera mucho más eficiente que las CPUs tradicionales. CUDA permite a los desarrolladores escribir y optimizar código paralelo de manera más intuitiva, haciendo uso de la jerarquía de hilos, bloques y cuadrículas para distribuir las tareas computacionales.

Desde su lanzamiento, CUDA ha evolucionado considerablemente, expandiendo sus capacidades y soporte de hardware, y se ha convertido en una herramienta fundamental en campos como la inteligencia artificial, el aprendizaje automático, la simulación científica, la visión por computadora y muchas otras áreas que requieren un alto rendimiento computacional.

5.1. Arquitectura de CUDA

La arquitectura CUDA se basa en un modelo de programación paralelo que permite a los desarrolladores aprovechar la capacidad de procesamiento masivo de las GPUs. Este modelo incluye varios componentes clave: el modelo de programación, la jerarquía de hilos, y los distintos tipos de memoria.

El modelo de programación de CUDA se organiza en torno a la ejecución de funciones denominadas kernels. Un kernel es una función que se ejecuta en la GPU y es llamada desde el host (la CPU). La ejecución de un kernel implica la creación de un gran número de hilos (threads), que se agrupan en bloques de hilos (blocks), y estos a su vez se agrupan en una cuadrícula de bloques (grid).

Los kernels se definen usando la palabra clave __global__ y se invocan desde el host utilizando la sintaxis <<< >>>, que especifica el número de bloques y el número de hilos por bloque. Al invocar un kernel, se debe especificar el número de bloques y el número de hilos por bloque utilizando la sintaxis <<<numBlocks, threadsPerBlock>>>.

5.1.1. Jerarquía de hilos

CUDA cuenta con una jerarquía de hilos la cual permite una organización estructurada y escalable de la ejecución paralela, permitiendo manejar eficientemente tanto pequeños como grandes conjuntos de datos. A continuación se explica dicha jerarquía la cual se puede ver de forma más gráfica en la figura 4

- **Hilos (Threads):** Son las unidades básicas de ejecución en CUDA. Cada hilo ejecuta una instancia del kernel y tiene su propio conjunto de registros y memoria local.
- **Bloques de Hilos (Blocks):** Los hilos se agrupan en bloques. Cada bloque tiene un tamaño fijo y puede ejecutar hasta 1024 hilos (en las arquitecturas actuales de NVIDIA). Los hilos dentro de un bloque pueden cooperar utilizando la memoria compartida y pueden sincronizarse utilizando las funciones de sincronización de hilos.
- **Cuadrícula de Bloques (Grid):** Los bloques se agrupan en una cuadrícula. La cuadrícula puede ser unidimensional, bidimensional o tridimensional, lo que permite una gran flexibilidad para organizar la ejecución paralela.

Además, CUDA nos proporciona la función *__syncthreads*(), la cual incroniza todos los hilos dentro de un bloque.

5.1.2. Memoria

En cuanto a la memoria, CUDA proporciona varios tipos de memoria, cada uno con sus propias características y usos específicos:

- **Memoria Global:** Es accesible por todos los hilos y tiene una alta capacidad pero también una latencia alta. Se utiliza para almacenar grandes volúmenes de datos que necesitan ser compartidos entre múltiples bloques.
- **Memoria Compartida:** Es accesible solo por los hilos dentro del mismo bloque y tiene una latencia mucho más baja que la memoria global. Se utiliza para la comunicación y cooperación entre hilos de un mismo bloque.

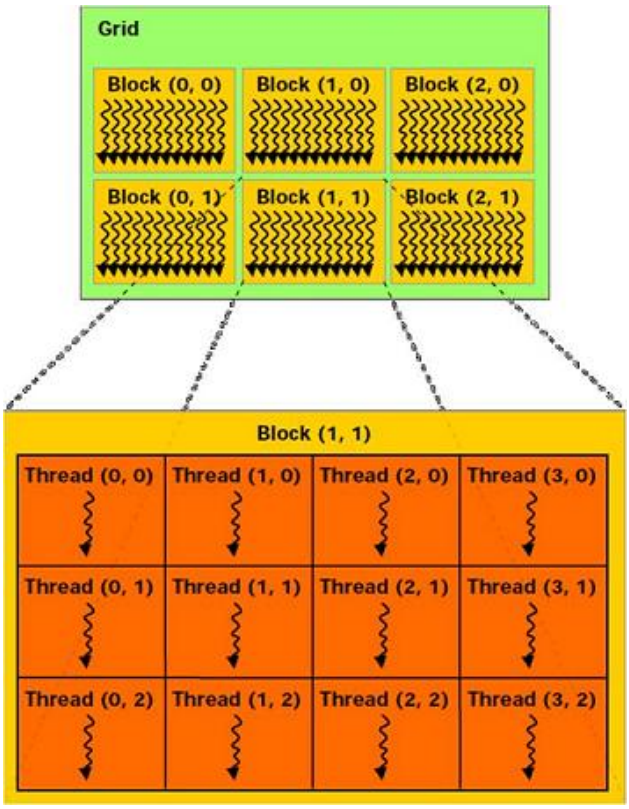

Figura 4: Jerarquía de hilos en CUDA.
[13]

- **Memoria Constante y de Textura:** Son tipos de memoria de solo lectura que pueden ser utilizados para almacenar datos constantes durante la ejecución del kernel. Tienen una latencia más baja cuando son accedidas.
- **Registros:** Se utilizan para almacenar variables locales de los hilos y tienen la latencia más baja. Sin embargo, el número de registros es limitado y su uso debe ser optimizado cuidadosamente.

Por otro lado, CUDA proporciona una variedad de funciones y librerías para gestionar la memoria:

- cudaMalloc: Asigna memoria en la GPU.
- cudaMemcpy: Copia datos entre la memoria del host y la memoria de la GPU.
- cudaFree: Libera la memoria asignada en la GPU.

5.2. Ventajas y aplicaciones

Una de las principales ventajas de CUDA es su capacidad para mejorar el rendimiento de las aplicaciones a través de la paralelización masiva. Las GPUs están

diseñadas con cientos o miles de núcleos de procesamiento que pueden ejecutar miles de hilos en paralelo. Esto permite a las aplicaciones que utilizan CUDA realizar grandes cantidades de cálculos simultáneamente, lo que resulta en una aceleración significativa en comparación con la ejecución en CPUs tradicionales.

Por otro lado, La arquitectura de CUDA es altamente escalable, lo que permite a los desarrolladores manejar problemas de diferentes tamaños sin una reestructuración significativa del código. La jerarquía de hilos, bloques y cuadrículas en CUDA facilita la escalabilidad, permitiendo que las aplicaciones se adapten fácilmente a diferentes tamaños de datos y configuraciones de hardware.

Tampoco nos podemos olvidar de la gran flexibilidad que proporciona CUDA a los desarrolladores, permitiéndoles integrar su código CUDA con una variedad de lenguajes de programación y herramientas. Esto facilita el uso de CUDA en diferentes contextos y aplicaciones. En este trabajo estamos trabajando con C++ pero esta tecnología también puede ser usada con Python, Fortran e innumerables lenguajes más.

Por último, CUDA ha demostrado ser extremadamente útil en una amplia variedad de campos y aplicaciones, lo que destaca su versatilidad y potencia. Tiene diversas aplicaciones en el campo de la ciencia y la tecnología, utilizándose para simulaciones numéricas, modelado molecular, análisis de datos, etc. Es de gran utilizad en la visión por computadora y procesamiento de imágenes puesto qu acelera tareas como el reconocimiento de objetos, análisis de imágenes médicas, y procesamiento de video. También es fundamental en la inteligencia artificial y el aprendizaje automático, donde se mejora el entrenamiento y la inferencia de modelos de redes neuronales profundas, permitiendo manejar grandes conjuntos de datos y realizar cálculos complejos de manera eficiente.

6. Implementación con CUDA

En esta sección se procede a explicar cómo se ha paralelizado el algoritmo DBC utilizando la tecnología CUDA. La implementación paralela con CUDA nos permite aprovechar la capacidad de procesamiento masivo de las GPUs para manejar de manera eficiente grandes volúmenes de datos y operaciones de cómputo.

La implementación secuencial del algoritmo DBC en C++ explicada en el capítulo 4 nos ha proporcionado una base sólida, pero a medida que aumentamos la complejidad y el tamaño de los datos, resulta evidente la necesidad de optimizar el rendimiento. El uso de CUDA nos permite dividir el trabajo entre miles de núcleos de procesamiento que pueden ejecutar hilos en paralelo, acelerando significativamente el tiempo de ejecución del algoritmo. En comparación con la implementación secuencial, la paralelización con CUDA no solo mejora el rendimiento, sino que también abre la puerta a analizar datos más complejos y de mayor tamaño en un tiempo razonable.

En este capítulo, se va a detallar cómo se ha adaptado y extendido el algoritmo DBC para ejecutarse en una GPU utilizando CUDA. Se procederá a explicar la estructura del código, las optimizaciones específicas que hemos implementado, y se analizará el impacto en el rendimiento a través de diversos experimentos.

6.1. Estructura del Código

En esta sección se procede a describir la estructura del código de la implementación paralela del algoritmo DBC utilizando CUDA. La implementación se ha diseñado para aprovechar al máximo las capacidades de procesamiento paralelo de las GPUs, dividiendo las tareas entre múltiples núcleos para lograr una ejecución más rápida y eficiente.

La implementación del algoritmo se divide en dos partes principales. La primera parte es la función DBCWithCuda, la cual se encarga de la configuración inicial, la asignación de memoria en la GPU, la transferencia de datos y la coordinación del lanzamiento de los kernels. Esta función actúa como el controlador principal que orquesta el flujo de datos y la ejecución del algoritmo en la GPU. La segunda parte comprende las funciones DBCKernel_inicial y DBCKernel, que son los kernels de CUDA donde se realiza el procesamiento intensivo. Estos kernels ejecutan el cálculo del DBC en paralelo, aprovechando la capacidad de procesamiento masivo de la GPU para obtener resultados de manera eficiente. Al dividir el algoritmo de esta manera, se facilita la gestión de los recursos y se optimiza el rendimiento del procesamiento paralelo.

6.1.1. DBCWithCuda

Como se puede observar en el listado 4, la función $DBCWithCuda$ es la encargada de ejecutar el algoritmo DBC utilizando la tecnología CUDA. A continuación, se detalla el funcionamiento de esta función, que realiza los pasos necesarios para configurar y ejecutar los kernels en la GPU.

Comenzando por las variables y el uso de memoria, se declaran los punteros dev_Imax, dev_Imin y dev_Nr, los cuales se van a utilizar para ir guardando los valores máximos, mínimos y soluciones respectivamente. Para estas variables utilizamos la función $cudaMalloc()$ para asignarles memoria en la GPU. Más adelante se utilizarán las funciones $cudaMemcpy$ para copiar los datos de la matriz $Imax$ desde la memoria del host a la memoria de la GPU y $cudaMemset$ para inicializar dev_Nr a cero. Por otro lado tenemos la variable $cudaStatus$, de tipo cudaError_t. Dicha variable se va a utilizar a lo largo de toda la ejecución del algoritmo para manejar y verificar el estado de las operaciones CUDA, permitiendo detectar posibles errores.

En cuanto a la configuración de la ejecución de los kernels se ha optado por una configuración de una dimensión. Con la orden $(block(TPB, 1, 1)$ se establecen TPB hilos en la dimensión x y un hilo en las dimensiones y y z. El valor de TPB (Threads Per Block) se ha establecido a 128 y se explicará en el apartado 6.2. El tamaño del grid se establece dinámicamente en el código según el número de bloques necesarios para cubrir todos los elementos a procesar y al igual que hemos hecho con block, se utilizará la dimensión $grid.x$, mientras que $grid.y$ y $grid.z$ se inicializan a 1.

Seguimos con la ejecución del kernel. Si nos fijamos, en el código mostrado en el listado 4 podemos observar que en la primera iteración del algoritmo se lanza la función $DBCKernel\_Inicial$ mientras que en las demás iteraciones se lanza $DBCKernel$. Esto es debido a que en la primera iteración, tanto para el cálculo de los valores máximos como los mínimos estamos sacando los datos de la matriz inicial, la cual se encuentra en $Imax$. En este paso guardaremos los valores máximos y mínimos de cada submatriz $s \times s \times s \times s$ en las matrices dev_Imax y dev_Imin respectivamente. Esto provoca que en las siguientes iteraciones ($DBCKernel$) los datos se lean de su matriz correspondiente, es decir, para calcular los valores máximos leemos de $Imax$ y para los valores mínimos leemos de $Imin$.

Una vez terminada la ejecución del cálculo del algoritmo se utiliza la función de CUDA $cudaDeviceSynchronize$ para asegurar que todos los kernels hayan terminado de ejecutarse antes de copiar los resultados de vuelta a la memoria del host.

Finalmente se liberan los recursos de memoria en la GPU utilizando cudaFree y se retorna el estado de CUDA.

```
1 DBCWithCuda(const unsigned char* Imax, const int Numr, const int M,
      const unsigned char bits_M, const int G, unsigned int* Nr)
2 {
```

```
3     unsigned char* dev_Imax = 0, dev_Imin = 0;
4     unsigned int* dev_Nr = 0;
5     cudaError_t cudaStatus;
6
7     cudaSetDevice(0);
8     cudaMalloc((void**)&dev_Imax, M*M*M*M*sizeof(unsigned char));
9     cudaMalloc((void**)&dev_Imin, M*M*M*M*sizeof(unsigned char));
10    cudaMalloc((void**)&dev_Nr, Numr * sizeof(unsigned int));
11
12    unsigned int num_box;
13    unsigned int tam_grid;
14    dim3 grid, block(TPB, 1, 1);
15
16    unsigned int sp, s = 2, sm, size = M, tpb;
17    unsigned char Nri = 0;
18    unsigned char b_tpb = log(TPB) / log(2);
19
20    cudaMemcpy(dev_Imax, Imax, M*M*M*M*sizeof(unsigned char),
          cudaMemcpyHostToDevice);
21    cudaStatus = cudaMemset(dev_Nr, 0, Numr*sizeof(unsigned int));
22
23    sm = s >> 1;
24    sp = ceilf(((float)(G << (Nri + 1))) / (float)M);
25    grid.x = ceilf(((M*M*M*M) / (s*s*s*s)) / (float)TPB);
26    grid.y = 1; grid.z = 1;
27
28    DBCKernel_inicial<<<grid, block>>>(dev_Imax, dev_Imin, M,
          bits_M, G, &dev_Nr[Nri], sm, sp, Nri + 1, b_tpb);
29
30    Nri++;
31    s <<= 1;
32    size >>= 1;
33
34    while (size > 2) {
35      sm = s >> 1;
36      sp = ceilf(((float)(G << (Nri + 1))) / (float)M);
37      num_box = (M * M * M * M) / (s * s * s * s);
38      if (num_box >= TPB) {
39        grid.x = ceilf(num_box / (float)TPB);
40        grid.y = 1; grid.z = 1;
41      }
42      else {
43        grid.x = 1; grid.y = 1; grid.z = 1;
44        block.x = num_box; block.y = 1; block.z = 1;
45        b_tpb = log(num_box) / log(2);
46      }
47      DBCKernel<<<grid, block>>>(dev_Imax, dev_Imin, M, bits_M, G,
```

```
            &dev_Nr[Nri], sm, sp, Nri + 1, b_tpb);
48        Nri++;
49        s <<= 1;
50        size >>= 1;
51      }
52      cudaDeviceSynchronize();
53      cudaMemcpy(Nr, dev_Nr, Numr*sizeof(unsigned int),
            cudaMemcpyDeviceToHost);
54
55    return cudaStatus;
56  }
```

Listing 4: Código CUDA para DBCWithCuda

6.1.2. DBCKernel

El kernel CUDA se utiliza para ejecutar la parte central del algoritmo DBC en la GPU. Como se ha comentado anteriormente, en la primera iteración se va a utilizar la función DBCKernel_Inicial, la cual es exactamente igual a esta con la única diferencia que al llamar a la función auxiliar $max_min_if()$ se va a pasar como parámetros $Imax$ tanto para el cálculo de máximos como para el cálculo de mínimos. En las siguientes iteraciones sí se va a utilizar DBCKernel, cuyo código se muestra en el listado 5. Esta función es la responsable de calcular los valores máximos y mínimos dentro de submatrices de tamaño $s \times s \times s \times s$ y actualizar el contador de cajas necesarias (Nr) utilizando operaciones atómicas. A continuación, se explican los componentes clave del kernel:

- **Identificación del thread y del índice global:** Cada hilo se identifica mediante threadIdx.x y su índice global se calcula combinando el índice del bloque y el índice del hilo dentro del bloque.
- **Cálculo de índices:** Se calculan los índices i, j, k y l para identificar las posiciones dentro de las submatrices.
- **Cálculo de desplazamientos:** Se calculan los desplazamientos necesarios para acceder a las posiciones correctas en las matrices Imax e Imin.
- **Cálculo de Valores Máximos y Mínimos:** Se utiliza la función max_min_if para calcular los valores máximos y mínimos dentro de la submatriz y se almacenan en las matrices $Imax$ e $Imin$.
- **Actualización del contador Nr:** Finalmente, se actualiza el contador Nr utilizando una operación atómica para asegurar la coherencia de los datos en un entorno de ejecución paralelo.

```
1  DBCKernel(unsigned char* Imax, unsigned char* Imin, const int M,
       const unsigned char bits_M, const int G, unsigned int* Nr, const
        unsigned int sm, const unsigned int sp, const unsigned char
       bits_s, const unsigned char bits_TPB)
2  {
3    register unsigned long long int tid=threadIdx.x;
4    register unsigned long long int idx=(blockIdx.x << bits_TPB)+tid;
5
6    register unsigned int l = idx >> ((bits_M - bits_s) + (bits_M -
         bits_s) + (bits_M - bits_s));
7    register unsigned int offsetl = (idx & (((M >> bits_s) << ((
         bits_M - bits_s) + (bits_M - bits_s))) - 1));
8    register unsigned int k = offsetl >> ((bits_M - bits_s) + (bits_M
          - bits_s));
9    register unsigned int offset = (offsetl & (((M >> bits_s) << (
         bits_M - bits_s)) - 1));
10   register unsigned long long int i = offset >> (bits_M - bits_s);
11   register unsigned long long int j = offset & ((M >> bits_s) - 1);
12
13   const register unsigned int jbs = j << bits_s;
14   const register unsigned int ibsbM = (i << bits_s) << bits_M;
15   const register unsigned int kbsbMM = ((k << bits_s) << bits_M) <<
          bits_M;
16   const register unsigned int lbsbMMM = (((l << bits_s) << bits_M)
         << bits_M) << bits_M;
17   const register unsigned int jbssm = (j << bits_s) + sm;
18   const register unsigned int ibssmbM = ((i << bits_s) + sm) <<
         bits_M;
19   const register unsigned int kbssmbMM = (((k << bits_s) + sm) <<
         bits_M) << bits_M;
20   const register unsigned int lbssmbMMM = ((((l << bits_s) + sm) <<
          bits_M) << bits_M) << bits_M;
21
22   register unsigned char valmax, valmin;
23   max_min_if(Imax[lbsbMMM + kbsbMM + ibsbM + jbs],
24              Imax[lbsbMMM + kbsbMM + ibsbM + jbssm],
25              Imax[lbsbMMM + kbsbMM + ibssmbM + jbs],
26              Imax[lbsbMMM + kbsbMM + ibssmbM + jbssm],
27              Imax[lbsbMMM + kbssmbMM + ibsbM + jbs],
28              Imax[lbsbMMM + kbssmbMM + ibsbM + jbssm],
29              Imax[lbsbMMM + kbssmbMM + ibssmbM + jbs],
30              Imax[lbsbMMM + kbssmbMM + ibssmbM + jbssm],
31              Imax[lbssmbMMM + kbsbMM + ibsbM + jbs],
32              Imax[lbssmbMMM + kbsbMM + ibsbM + jbssm],
33              Imax[lbssmbMMM + kbsbMM + ibssmbM + jbs],
34              Imax[lbssmbMMM + kbsbMM + ibssmbM + jbssm],
```

```
35              Imax[lbssmbMMM + kbssmbMM + ibsbM + jbs],
36              Imax[lbssmbMMM + kbssmbMM + ibsbM + jbssm],
37              Imax[lbssmbMMM + kbssmbMM + ibssmbM + jbs],
38              Imax[lbssmbMMM + kbssmbMM + ibssmbM + jbssm],
39
40              Imin[lbsbMMM + kbsbMM + ibsbM + jbs],
41              Imin[lbsbMMM + kbsbMM + ibsbM + jbssm],
42              Imin[lbsbMMM + kbsbMM + ibssmbM + jbs],
43              Imin[lbsbMMM + kbsbMM + ibssmbM + jbssm],
44              Imin[lbsbMMM + kbssmbMM + ibsbM + jbs],
45              Imin[lbsbMMM + kbssmbMM + ibsbM + jbssm],
46              Imin[lbsbMMM + kbssmbMM + ibssmbM + jbs],
47              Imin[lbsbMMM + kbssmbMM + ibssmbM + jbssm],
48              Imin[lbssmbMMM + kbsbMM + ibsbM + jbs],
49              Imin[lbssmbMMM + kbsbMM + ibsbM + jbssm],
50              Imin[lbssmbMMM + kbsbMM + ibssmbM + jbs],
51              Imin[lbssmbMMM + kbsbMM + ibssmbM + jbssm],
52              Imin[lbssmbMMM + kbssmbMM + ibsbM + jbs],
53              Imin[lbssmbMMM + kbssmbMM + ibsbM + jbssm],
54              Imin[lbssmbMMM + kbssmbMM + ibssmbM + jbs],
55              Imin[lbssmbMMM + kbssmbMM + ibssmbM + jbssm],
56              valmax, valmin);
57
58    Imax[lbsbMMM + kbsbMM + ibsbM + jbs] = valmax;
59    Imin[lbsbMMM + kbsbMM + ibsbM + jbs] = valmin;
60
61    float invsp = 1.0 / sp;
62    atomicAdd(Nr, ceilf((float) /valmax * invsp) - ceilf((float) /
          valmin * invsp) + 1);
63  }
```

Listing 5: Código CUDA para DBCKernel

6.2. Selección del número de hilos por bloque (TPB)

Una de las decisiones cruciales al implementar algoritmos en CUDA es la selección del tamaño de los bloques de hilos, conocido como TPB (Threads Per Block) [14]. Este valor tiene un impacto significativo en el rendimiento del algoritmo, ya que determina cómo se distribuyen los hilos entre los bloques y cómo se accede a la memoria.

El valor de TPB óptimo depende de varios factores, incluyendo la arquitectura específica de la GPU utilizada y la naturaleza del algoritmo en cuestión. En general, un TPB de 128 hilos por bloque es una configuración comúnmente recomendada en la documentación de CUDA, y existen varias razones para esta elección.

Primero, utilizar un TPB de 128 hilos por bloque permite una buena utilización

de los recursos de la GPU. Las GPUs están diseñadas para ejecutar múltiples hilos en paralelo, y un tamaño de bloque de 128 hilos suele ser suficiente para ocupar los recursos de manera eficiente sin incurrir en la sobrecarga de sincronización y gestión de bloques más grandes [15]. Esta configuración equilibra la carga de trabajo entre los bloques y los multiprocesadores de la GPU, maximizando el rendimiento sin comprometer la simplicidad de la gestión de hilos.

Además, un TPB de 128 hilos permite un acceso más eficiente a la memoria compartida y global. La memoria compartida en CUDA es una memoria de alta velocidad que puede ser utilizada por todos los hilos de un bloque. Con 128 hilos por bloque, se puede organizar y acceder a esta memoria de manera más efectiva, reduciendo las latencias y mejorando el rendimiento global del algoritmo.

Finalmente, la elección de 128 hilos por bloque facilita la optimización y el ajuste del rendimiento del algoritmo. Este tamaño de bloque es suficientemente flexible para ser ajustado en diferentes configuraciones de hardware sin requerir cambios significativos en el código. Además, permite una escalabilidad adecuada, asegurando que el algoritmo pueda ser ejecutado de manera eficiente en una variedad de GPU con diferentes capacidades de multiprocesamiento.

7. Estudio comparativo de tiempos de ejecución

En este capítulo se presenta un análisis comparativo de los tiempos de ejecución entre la implementación secuencial y la implementación paralela en CUDA del algoritmo DBC. El objetivo principal es evaluar la eficiencia y escalabilidad de cada versión del algoritmo, determinando así las ventajas y limitaciones de utilizar un enfoque paralelo para el procesamiento de grandes volúmenes de datos.

Este estudio no solo compara los tiempos de ejecución de las versiones secuencial y paralela del algoritmo en el PC del alumno, sino que también incluye la ejecución de la versión paralela en un servidor de alto rendimiento. Este enfoque permite evaluar la escalabilidad de la implementación paralela y su comportamiento en un entorno de computación más potente.

El análisis se realiza utilizando diferentes tamaños de matriz, lo que permite observar cómo varía la eficiencia en función del tamaño de los datos. Además, se presentan tablas y gráficos comparativos para ilustrar claramente las diferencias de rendimiento entre las dos versiones del algoritmo y su ejecución en diferentes entornos.

7.1. Metodología empleada

En este apartado se describen los métodos y procedimientos utilizados para realizar el análisis comparativo de los tiempos de ejecución entre la implementación secuencial y la implementación paralela del algoritmo DBC. El objetivo es proporcionar una evaluación precisa y reproducible de la eficiencia y escalabilidad de ambas versiones del algoritmo.

7.1.1. Entornos de prueba

Para llevar a cabo las pruebas, se utilizaron dos entornos distintos: el portátil personal del alumno y un servidor de supercomputación Isicomputing.ugr.es, perteneciente al Departamento de Lenguajes y Sistemas Informáticos de la UGR.. Las especificaciones detalladas del hardware utilizado en ambos entornos se encuentran en la Sección 3 de este documento. A continuación, se resume brevemente cada entorno:

- **Entorno local:**
  - Procesador: AMD Ryzen 7 4800H
  - Memoria RAM: 16GB
  - GPU: NVIDIA GeForce RTX 3060, 3584 CUDA cores, 12GB de memoria.

- **Servidor:**
  - Procesador: Intel Xeon Silver 4210 2.20GHz
  - Memoria RAM: 96GB
  - GPU: NVIDIA Tesla V100 PCIexpress, 5120 CUDA cores, 32GB de memoria.

7.1.2. Parámetros de configuración

Para garantizar la consistencia y comparabilidad de los resultados, se utilizaron los siguientes parámetros de configuración para todas las ejecuciones del algoritmo:

- **Tamaños de entrada**: Se utilizaron matrices de tamaños incrementales para evaluar el rendimiento del algoritmo en distintos volúmenes de datos: 8x8x8x8, 16x16x16x16, 32x32x32x32, 64x64x64x64, y 128x128x128x128.
- **Número de repeticiones:** Cada ejecución del algoritmo se repitió 10 veces para obtener una medida precisa del tiempo de ejecución promedio y reducir el impacto de posibles fluctuaciones en el rendimiento del hardware.

7.1.3. Procedimiento

El procedimiento seguido para la ejecución de las pruebas es el siguiente:

- **Implementación secuencial:**
  - Se ejecutó la versión secuencial del algoritmo DBC para cada tamaño de entrada en el PC.
  - Se registraron los tiempos de ejecución para cada repetición y se calculó el tiempo de ejecución promedio.
- **Implementación paralela en CUDA:**
  - Se ejecutó la versión paralela del algoritmo DBC en CUDA para cada tamaño de entrada en el PC.
  - Se registraron los tiempos de ejecución para cada repetición y se calculó el tiempo de ejecución promedio.
  - Posteriormente, se replicó el mismo proceso en el servidor de alto rendimiento para evaluar la escalabilidad y eficiencia del algoritmo en un entorno más potente.
- **Análisis de datos:**

- Se compararon los tiempos de ejecución obtenidos para las versiones secuencial y paralela del algoritmo en ambos entornos.
- Se utilizaron tablas y gráficos para visualizar las diferencias de rendimiento y facilitar el análisis comparativo.

Al seguir esta metodología, se garantiza una evaluación exhaustiva y precisa de la eficiencia y escalabilidad de las implementaciones secuencial y paralela del algoritmo DBC, proporcionando una base sólida para comparar su rendimiento y determinar las ventajas del procesamiento paralelo en aplicaciones de gran escala.

7.2. Resultados de tiempos de ejecución

En este apartado se presentan los resultados obtenidos de las pruebas de tiempos de ejecución realizadas para las implementaciones secuencial y paralela del algoritmo DBC. Los resultados se han obtenido tanto en el entorno local como en un servidor de alto rendimiento, lo que nos permite evaluar la eficiencia de ambas implementaciones en diferentes escenarios de hardware.

Las Tablas 1 y 2 muestran los tiempos de ejecución (en milisegundos) para diferentes tamaños de matriz y los factores de aceleración obtenidos al comparar las ejecuciones en GPU con la ejecución en CPU. Los factores de aceleración se calculan como la relación entre el tiempo de ejecución en CPU y el tiempo de ejecución en GPU.

La Tabla 1 presenta los resultados obtenidos en un PC local. Como se puede observar, el tiempo de ejecución en la CPU aumenta significativamente con el tamaño de la imagen, mientras que el uso de la GPU mejora considerablemente los tiempos de ejecución, especialmente para imágenes de mayor tamaño.

	CPU	GPU	
Tamaño de la imagen	Tiempo	Tiempo	Aceleración
8^4	1.0	2.1	0.48
16^4	2.0	2.3	0.87
32^4	39.0	4.2	9.29
64^4	522.0	28.6	18.25
128^4	6889.0	420.2	16.39

Cuadro 1: Comparación de tiempos de procesamiento entre CPU y GPU en PC.

La Tabla 2 muestra los resultados obtenidos en el servidor de alto rendimiento. Aquí, los tiempos de ejecución son notablemente menores en comparación con el entorno local, gracias a la mayor capacidad de procesamiento del servidor. La aceleración lograda con la GPU del servidor es también significativamente mayor, destacando la eficiencia del algoritmo paralelizado en un entorno más potente.

Tamaño de la imagen	CPU servidor	GPU servidor	
	Tiempo	Tiempo	Aceleración
8^4	0.46	0.33	1.39
16^4	8.99	0.43	20.91
32^4	48.53	1.77	27.42
64^4	504.83	22.18	22.76
128^4	6937.11	361.84	19.17

Cuadro 2: Comparación de tiempos de procesamiento entre CPU y GPU en servidor.

Estos resultados demuestran la eficiencia y ventaja significativa de utilizar GPUs para la implementación del algoritmo DBC, especialmente en entornos de alto rendimiento, lo que permite manejar grandes volúmenes de datos de manera más eficiente y rápida. En el siguiente apartado se analizará en detalle la eficiencia y escalabilidad de ambas implementaciones.

7.3. Análisis de resultados

En este apartado, se analiza detalladamente el rendimiento de las implementaciones secuencial y paralela del algoritmo DBC mediante el examen de los tiempos de ejecución obtenidos en el entorno local y en el servidor de alto rendimiento. Se utiliza el gráfico comparativo de la Figura 5 para visualizar las diferencias de rendimiento y se discuten las implicaciones de estos resultados en términos de eficiencia y escalabilidad.

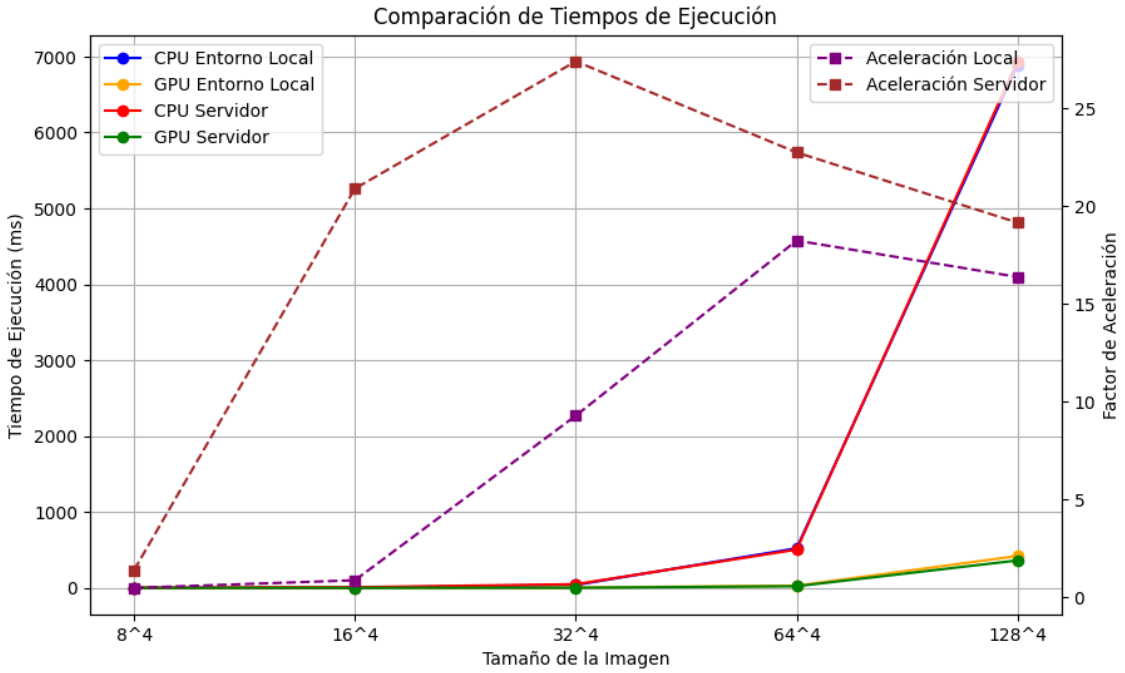

Figura 5: Comparación de tiempos de ejecución para diferentes tamaños de imagen entre CPU y GPU.

Los tiempos de ejecución en la CPU del PC aumentan exponencialmente con el tamaño de la imagen. Para la imagen de mayor tamaño (128^4), el tiempo de ejecución alcanza los 6889 milisegundos. Esto refleja la alta complejidad computacional y las limitaciones de procesamiento cuando se utiliza una implementación secuencial.

La implementación en GPU del entorno local muestra una mejora significativa en los tiempos de ejecución en comparación con la CPU. Sin embargo, para la imagen más pequeña (8^4), el tiempo de ejecución en la GPU (2.1 segundos) es mayor que en la CPU debido a la sobrecarga asociada a la inicialización y transferencia de datos a la GPU. A medida que el tamaño de la imagen aumenta, la GPU demuestra su ventaja, con tiempos de ejecución mucho menores. Para la imagen de 128^4, el tiempo de ejecución es de 420.2 segundos, lo que representa una aceleración de 16.51 veces en comparación con la CPU.

Los tiempos de ejecución en la GPU del servidor son consistentemente menores que en la GPU del entorno local. Para la imagen de mayor tamaño, el tiempo de ejecución se reduce a 335.71 segundos. La aceleración obtenida al utilizar la GPU es notable. En el entorno local obtenemos una aceleración media de 9.56. En el servidor la aceleración es aún mayor, obteniendo una aceleración media de 18.33. Esta mejora se debe a la mayor capacidad y rendimiento del hardware del servidor, que incluye una GPU más potente.

7.3.1. Eficiencia de la implementación paralela

El análisis de los resultados muestra que la implementación paralela en CUDA ofrece ventajas significativas en términos de eficiencia, especialmente para grandes volúmenes de datos. Las GPUs están diseñadas para manejar múltiples operaciones en paralelo, lo que les permite procesar grandes conjuntos de datos de manera mucho más rápida que las CPUs tradicionales.

- Sobrecarga Inicial y Transferencia de Datos: Aunque la GPU muestra un rendimiento inferior para los datos más pequeños debido al overhead inicial y a la necesidad de transferir datos entre la memoria del host y la memoria de la GPU, este efecto se desaparece rápidamente a medida que aumenta el tamaño de la entrada.

- Escalabilidad: La implementación en CUDA muestra una excelente escalabilidad. A medida que el tamaño de la entrada aumenta, la aceleración proporcionada por la GPU también aumenta, lo que sugiere que la implementación paralela es altamente eficiente para aplicaciones que requieren el procesamiento de grandes volúmenes de datos.

7.3.2. Impacto hardware

El hardware utilizado tiene un impacto significativo en los tiempos de ejecución y la eficiencia del algoritmo. El servidor de alto rendimiento, equipado con una GPU NVIDIA Tesla V100, muestra una mejora considerable en comparación con el entorno local, lo que destaca la importancia de utilizar hardware adecuado para maximizar las capacidades de procesamiento paralelo. Las GPUs más avanzadas y la mayor cantidad de memoria en el servidor permiten manejar conjuntos de datos más grandes de manera más eficiente.

8. Aplicación del algoritmo en casos reales

En este capítulo se procede a aplicar el algoritmo DBC en 4D sobre datos reales de electroencefalografía (EEG) [16] para identificar posibles diferencias y patrones relevantes en diferentes estados de conciencia. El objetivo es mejorar los datos obtenidos por el profesor Juan Ruiz de Miras con el algoritmo Box-Counting [17]. Dicho algoritmo solo cuenta ceros y unos, a diferencia del DBC que calcula la diferencia entre diferentes niveles de gris que van desde 0 a 255. Los datos analizados provienen de personas que se encuentran despiertas o dormidas artificialmente usando los anestésicos propofol y xenón. Para llevarlo a cabo vamos a utilizar la versión del algoritmo que hemos implementado con CUDA para poder aprovechar la capacidad de procesamiento masivo que nos proporcionan las unidades gráficas.

8.1. Obtención de matrices 4D a partir de EEG

La obtención de matrices 4D a partir de los datos de electroencefalografía (EEG) es un proceso que involucra varias etapas desde la adquisición de los datos hasta su preparación para el análisis de la dimensión fractal. A continuación, se describe el proceso.

1. **Adquisición de Datos EEG y TMS:** Se utilizan EEG de alta densidad (hd-EEG) combinados con estimulación magnética transcraneal (TMS) en sujetos sanos. Los datos se registran durante diferentes estados de conciencia: vigilia y sedación inducida por xenón y propofol. La TMS se aplica en áreas específicas del cerebro y se registra la respuesta cortical utilizando un amplificador EEG compatible con TMS y una bobina de estimulación guiada por resonancia magnética.

2. **Modelado de Fuentes Corticales:** Se modela el volumen conductor de la cabeza utilizando el método de las tres esferas concéntricas[18] para representar el cráneo interno, el cráneo externo y el cuero cabelludo. Esto se realiza usando el paquete de software Brainstorm [19]. La solución espacial se restringe a la corteza cerebral, que se modela como una malla de 3004 dipolos orientados perpendicularmente a la superficie cortical. Este modelo se adapta a la anatomía de cada sujeto usando el software SPM (Statistical Parametric Mapping) [20].

3. **Conversión de resultados a matrices 4D:** Los resultados obtenidos se organizan en una matriz tridimensional, donde los valores oscilan entre 0 y 255, representando distintos niveles que la activación cortical. La cuarta dimensión de la matriz corresponde al tiempo, capturando la evolución temporal

de la actividad cerebral. De esta manera, se generan matrices de dimensiones $128 \times 128 \times 128 \times 128$, donde las primeras tres dimensiones representan el espacio tridimensional de la coteza cerebral y la cuarta dimensión captura las diferentes muestras de tiempo durante el período de análisis. Esta estructura 4D permite un análisis detallado y preciso de los patrones espaciotemporales de activación cerebral inducidos por la TMS, proporcionando una base sólida para la posterior aplicación del algoritmo DBC.

8.2. Resultados

En este subapartado, se presentan los resultados obtenidos al aplicar el algoritmo DBC 4D implementado en CUDA en datos reales de electroencefalografía (EEG). El algoritmo DBC se aplicó a las matrices 4D obtenidas, las cuales representan la activación cortical en el espacio y en el tiempo. Este algoritmo calcula la dimensión fractal 4D (4DFD) de las matrices, proporcionando una medida cuantitativa de la complejidad de los patrones de activación cortical en diferentes estados de conciencia.

Los resultados del experimento se resumen en el gráfico 6. En dicho gráfico se presentan las distribuciones de la 4DFD para los tres estados de conciencia analizados: wake, xenón y propofol.

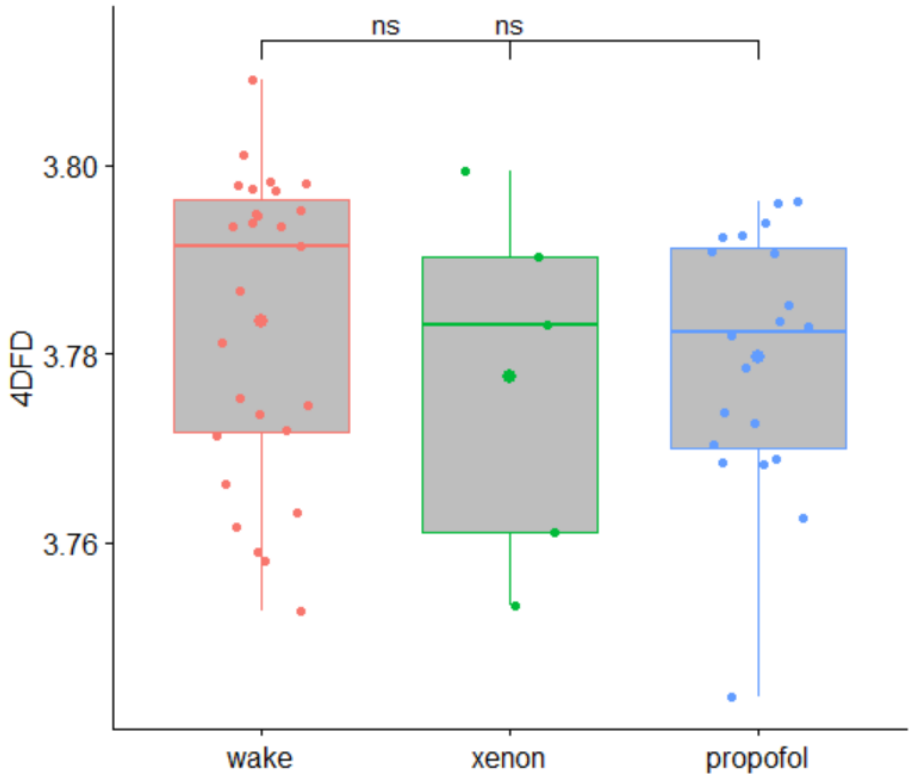

Figura 6: Distribuciones de la 4DFD para diferentes estados de conciencia.
ns: diferencia no significativa.

El gráfico muestra que la media de la 4DFD para los estados de sedación induci-

dos por xenón y propofol es menor en comparación con la media de la 4DFD cuando los sujetos están despiertos (wake). Esto sugiere que la complejidad de los patrones de activación cortical es menor durante la sedación, lo que podría estar relacionado con los efectos de los anestésicos en la actividad cerebral.

Para evaluar si la significancia de estas diferencias obervadas es suficiente para poder afirmar que el algoritmo DBC realiza una buena clasificación con los datos de prueba se va a realizar un análisis estadístico utilizando la prueba de Kruskal-Wallis[21] para muestras independientes. Esta prueba no paramétrica se utiliza debido a la naturaleza de los datos y la necesidad de comparar más de dos grupos.

La prueba de Kruskal-Wallis es una técnica no paramétrica que se utiliza para determinar si existen diferencias significativas entre las distribuciones de tres o más grupos independientes. Es una extensión de la prueba de Mann-Whitney U y es adecuada para datos que no necesariamente siguen una distribución normal. El procedimiento de la prueba es el siguiente:

1. Ranqueo de los Datos: Todos los valores de las muestras se combinan y se les asigna un rango, desde el menor al mayor.

2. Cálculo del Estadístico de Prueba H: Se calcula el estadístico H, que se basa en la suma de los rangos de cada grupo y el número de observaciones en cada grupo.

3. Comparación con la Distribución Chi-Cuadrado: El valor de H se compara con la distribución chi-cuadrado para determinar la significancia de las diferencias observadas.

En nuestro análisis, el resultado de la prueba de Kruskal-Wallis es $H = 1.34$, con un p-valor de 0.442. Esto indica que no existen diferencias significativas entre las distribuciones de la dimensión fractal dada por el algoritmo DBC para los tres estados de conciencia analizados (wake, xenón y propofol).

En conclusión, los resultados obtenidos indican que, aunque existe una tendencia observable en las diferencias de la 4DFD entre los estados de vigilia y sedación, estas diferencias no son estadísticamente significativas según la prueba de Kruskal-Wallis. Los resultados obtenidos sugieren que la dimensión fractal para datos en 4 dimensiones puede no ser suficiente por sí sola para discriminar de manera concluyente entre diferentes estados de conciencia.

Este análisis pone de manifiesto la necesidad de seguir investigando y de considerar otros enfoques complementarios para una evaluación más completa de los estados de conciencia a partir de datos de EEG. El uso del algoritmo DBC implementado en CUDA ha demostrado ser una herramienta eficiente y valiosa para este

tipo de estudios, permitiendo manejar y analizar grandes volúmenes de datos de manera efectiva.

9. Conclusiones

En este proyecto se ha trabajado la implementación del algoritmo Differential Box-Counting (DBC) para calcular la dimensión fractal en matrices de cuatro dimensiones, utilizando la tecnología CUDA para aprovechar el procesamiento paralelo de las tarjetas gráficas. Se ha realizado desde el estudio teórico del algoritmo hasta su implementación práctica y aplicación en datos reales de electroencefalografía (EEG), además de un estudio sobre la mejora en rendimiento respecto a una versión secuencial.

Se ha desarrollado una versión secuencial del algoritmo DBC en C++, que ha servido de base para la implementación paralela en CUDA. La versión paralela del algoritmo, ejecutada en GPUs, ha demostrado ser significativamente más rápida. El análisis comparativo de los tiempos de ejecución entre las versiones secuencial y paralela del algoritmo deja claro que la implementación en CUDA es mucho más eficiente y necesaria cuando se trata de grandes volúmenes de datos, como matrices de $18 \times 18 \times 18 \times 18$. En mi PC personal, se ha llegado a obtener una aceleración de 18.25, siendo la aceleración media de 9.56. Respecto a la ejecución en el servidor, la aceleración máxima obtenida ha sido de 27.42, siendo la aceleración media 18.33.La versión paralela, especialmente cuando se ejecutó en un servidor de alto rendimiento, fue considerablemente más rápida. Esto demuestra la capacidad del algoritmo para escalar con el uso de GPUs y subraya la importancia de contar con el hardware adecuado para maximizar las ventajas del procesamiento paralelo en aplicaciones de gran escala. Este avance es crucial para manejar y procesar datos complejos de manera efectiva y en tiempos razonables, mostrando el verdadero potencial del procesamiento paralelo.

Se ha aplicado el algoritmo DBC a datos reales de EEG obtenidos de sujetos en diferentes estados de conciencia: despiertos, sedado con propofol y sedados con xenón. Los resultados muestran una tendencia hacia una menor dimensión fractal en los patrones de activación cortical durante los estados de sedación en comparación con la vigilia. Sin embargo, estas diferencias no son estadísticamente significativas según la prueba de Kruskal-Wallis. Aunque observamos que la dimensión fractal 4D es una herramienta valiosa, por sí sola puede no ser suficiente para clasificar claramente entre diferentes estados de conciencia inducidos por anestésicos.

A pesar de que los resultados en la aplicación en casos reales no muestran diferencias significativas entre los diferentes estados de conciencia, este trabajo ha sentado una base sólida para futuras investigaciones. La eficiencia y la capacidad para manejar grandes volúmenes de datos demostradas por el algoritmo DBC implementado en CUDA ofrecen una herramienta prometedora para el análisis de EEG y otros tipos de datos biomédicos.

En conclusión, este trabajo ha demostrado la viabilidad y eficiencia del uso del algoritmo Differential Box-Counting implementado en CUDA para el análisis de datos en cuatro dimensiones. Aunque la aplicación en casos reales no ha sido concluyentes en términos de clasificación de estados de conciencia, el trabajo realizado proporciona una base sólida para futuras investigaciones y aplicaciones en este campo. La capacidad de procesamiento paralelo de las GPUs representa una herramienta poderosa para el análisis de grandes volúmenes de datos, abriendo nuevas posibilidades para avanzar en la comprensión de la actividad cerebral y los estados de conciencia.

10. Material entregado

En este último capítulo se detallan los documentos y archivos que se van a entregar como parte de este proyecto, así como los requisitos necesarios para su ejecución.

- **Memoria del proyecto** (este documento), que contiene a descripción detallda del trabajo realizado.
- **Carpeta con código fuente.** Esta carpeta se estructura de la siguiente manera:
  - 'common': Subcarpeta que contiene los ficheros necesarios para la lectura de archivos .vox y la toma de tiempos.
  - 'imagenes': Subcarpeta donde se guardan los archivos .vox con los datos de prueba.
  - DBC_4D_CPU: Subcarpeta que contiene la implementación secuencial del algoritmo.
  - DBC_4D_CPU: Subcarpeta que contiene la implementación paralela del algoritmo.

Para ejecutar el proyecto, es imprescindible contar con una tarjeta gráfica NVIDIA que soporte CUDA. Además se deberá tener instalada la versión adecuada del Toolkit de CUDA y los controladores correspondientes para su tarjeta gráfica NVIDIA.

Referencias

[1] Wikipedia contributors. *Dimensión fractal — Wikipedia, La enciclopedia libre*. 2024. URL: https://es.wikipedia.org/wiki/Dimensi%C3%B3n_fractal.

[2] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman y Company, 1983. URL: https://www.jstor.org/stable/2323761.

[3] Ayan Seal Chinmaya Panigrahy. *Differential box counting methods for estimating fractal dimension of gray-scale images: A survey*. 2019. URL: https://www.sciencedirect.com/science/article/abs/pii/S096007791930219X.

[4] S. Esakkiammal V. Lakshmi Praba. *Texture Analysis in Images with Differential Box Counting Algorithm*. 2019. URL: https://www.ijcseonline.org/spl_pub_paper/18-Final%20paper%20%20DIP%2021.pdf.

[5] Xiaokui Yue. *Feature extraction algorithm for space targets based on fractal theory*. 2007. URL: https://www.researchgate.net/publication/253900001_Feature_extraction_algorithm_for_space_targets_based_on_fractal_theory.

[6] Chen KS Tzeng YC Chen D. *Integration of spatial chaotic model and type-2 fuzzy sets to coastline detection in SAR images*. 2008.

[7] J. R. de Miras. «Fast differential box-counting algorithm on gpu». En: *The Journal of Supercomputing* 76.1 (2020), págs. 204-225.

[8] NVIDIA. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[9] Universidad de Granada. *Guía docente del trabajo fin de carrera*. URL: https://grados.ugr.es/informatica/pages/infoacademica/guias_docentes/201516/cuarto/2semestre/pfg/guiapfg.

[10] N. Sarkar B.B. Chaudhuri. *Texture segmentation using fractal dimension*. URL: https://ieeexplore.ieee.org/abstract/document/368149.

[11] Universidad de Sevilla. *Análisis de la complejidad de los algoritmos*. URL: https://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-28.html.

[12] *Documentación C++, Cplusplus*. URL: https://cplusplus.com/doc/.

[13] *Thread Hierarchy in CUDA Programming*. URL: https://cuda-programming.blogspot.com/2012/12/thread-hierarchy-in-cuda-programming.html.

[14] Wikipedia contributors. *Thread block (CUDA programming) — Wikipedia, La enciclopedia libre*. URL: https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming).

[15] NVIDIA Corporation. *CUDA C Programming Guide*. Versión 11.0. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[16] Ole Jensen, John E. Richards y Michael X. Cohen. «Introduction to EEG». En: *Chapter in "An Introduction to the Event-Related Potential Technique"* (2011). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3181835/.

[17] J. Ruiz de Miras et al. «Fractal dimension analysis of states of consciousness and unconsciousness using transcranial magnetic stimulation». En: *Computer Methods and Programs in Biomedicine* 175 (2019), págs. 129-137. DOI: 10.1016/j.cmpb.2019.04.017.

[18] P. Berg y M. Scherg. «A fast method for forward computation of multiple-shell spherical head models». En: *Electroencephalogr. Clin. Neurophysiol.* 90 (1994), págs. 58-64. DOI: 10.1016/0013-4694(94)90113-9.

[19] Brainstorm. *Brainstorm: Electrophysiology, MEG, EEG, ECoG, SEEG, DBS*. https://neuroimage.usc.edu/brainstorm/. Accessed: 2024-06-21. 2024.

[20] Statistical Parametric Mapping. *Statistical Parametric Mapping: SPM*. https://www.fil.ion.ucl.ac.uk/spm/. Accessed: 2024-06-21. 2024.

[21] DATAtab. *Prueba de Kruskal-Wallis*. URL: https://datatab.es/tutorial/kruskal-wallis-test.