Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Fast computation of fractal dimension for 2D, 3D and 4D data

J. Ruiz de Miras [a,*], M.A. Posadas [a], A.J. Ibáñez-Molina [b], M.F. Soriano [c], S. Iglesias-Parro [b]

[a] *Software Engineering Department, University of Granada, Granada, Spain*
[b] *Department of Psychology, University of Jaén, Jaén, Spain*
[c] *St. Agustín University Hospital, Linares, Jaen, Spain*

A R T I C L E I N F O

Keywords:
Fractal dimension
Box counting
GPU
Schizophrenia
EEG

A B S T R A C T

The box-counting (BC) algorithm is one of the most popular methods for calculating the fractal dimension (FD) of binary data. FD analysis has many important applications in the biomedical field, such as cancer detection from 2D computed axial tomography images, Alzheimer's disease diagnosis from magnetic resonance 3D volumetric data, and consciousness states characterization based on 4D data extracted from electroencephalography (EEG) signals, among many others. Currently, these kinds of applications use data whose size and amount can be very large, with high computation times needed to calculate the BC of the whole datasets. In this study we present a very efficient parallel implementation of the BC algorithm for its execution on Graphics Processing Units (GPU). Our algorithm can process 2D, 3D and 4D data and we tested it on two platforms with different hardware configurations. The results showed speedups of up to 92.38 $\times$ (2D), 57.27 $\times$ (3D) and 75.73 $\times$ (4D) with respect to the corresponding CPU single-thread implementations of the same algorithm. Against an OpenMP multi-thread CPU implementation, our GPU algorithm achieved speedups of up to 16.12 $\times$ (2D), 6.86 $\times$ (3D) and 7.49 $\times$ (4D). We have also compared our algorithm to a previous GPU implementation of the BC algorithm in 3D, achieving a speedup of up to 4.79 $\times$. Finally, as a practical application of our GPU BC algorithm a study comparing the FD of 4D data extracted from the EEGs of a schizophrenia patient and a healthy subject was performed. The computation time for processing 40 4D matrices was reduced from three hours (sequential CPU) to less than three minutes with our GPU algorithm.

1. Introduction and background

The box-counting (BC) algorithm [1] is one of the most-widely used methods for estimating the fractal dimension (FD) of a signal. Computing the FD [2] has relevant applications in many areas, especially in the biomedical field. The FD has been successfully used for diagnosing cancer [3,4], characterizing brain atrophy associated with several neurodegenerative diseases [5,6], and determining consciousness and unconsciousness states in sleeping and sedated subjects [7], among other biomedical applications. These applications are characterized by using large and multidimensional data such as 2D computed axial tomography images, 3D volumetric data from magnetic resonance imaging and 4D data extracted from electroencephalography recordings, respectively.

Basically, the FD of a binarized image (a typical example in 2D) can be computed as the least-square linear fit of $\log(n(s))$ against $\log(1/s)$, where $n(s)$ is the BC for the grid of size s. The BC of the grid of size s is computed as the number of boxes of size s that contain any non-zero pixels of the image. Equivalent definitions can be made for volumes in 3D and 4D (the fourth dimension is time), considering grids and boxes in the respective dimensions.

The computational cost of the BC algorithm when applied on large datasets of 2D, 3D or 4D data can be very high [8–10]. Moreover, several FD applications require real-time processing such as object recognition [11,12] and diagnosis based on time-dependent biomedical data [13]. The need for improving the computation cost of the BC algorithm appeared more than two decades ago [14], and this need is now even more crucial in many current applications and research areas.

To the best of our knowledge, only a few studies have provided efficient implementations of the BC algorithm. Hou et al. [15] developed a BC algorithm based on identifying each box through the binary representation of its coordinates. Then a sorted list with this binary representation of all non-zero boxes is used to rapidly identify which boxes are covered in each grid of size r. This algorithm, although efficient, was designed for executing in a sequential mode, with some other improvements such as the one presented in [16]. A fast parallel GPU

* Correspondence to: Software Engineering Department, Periodista Manuel Saucedo Aranda s/n, 18071 Granada, Spain.
E-mail address: demiras@ugr.es (J. Ruiz de Miras).

https://doi.org/10.1016/j.jocs.2022.101908
Received 7 July 2022; Received in revised form 13 October 2022; Accepted 10 November 2022
Available online 12 November 2022

implementation of this algorithm was provided by Jiménez et al. in [17]. This parallel BC algorithm was implemented in CUDA (Compute Unified Device Architecture) [18] and achieved an average speedup of 28 × regarding the CPU version of the algorithm.

In this study we also present a fast GPU implementation of the BC algorithm using CUDA. Our approach is based on a kernel that is able to compute and combine the BC of each individual grid in a very simple and efficient way in parallel. This fact allowed us to obtain particularized fast implementations of the BC algorithm for computing the FD of 2D, 3D and 4D data.

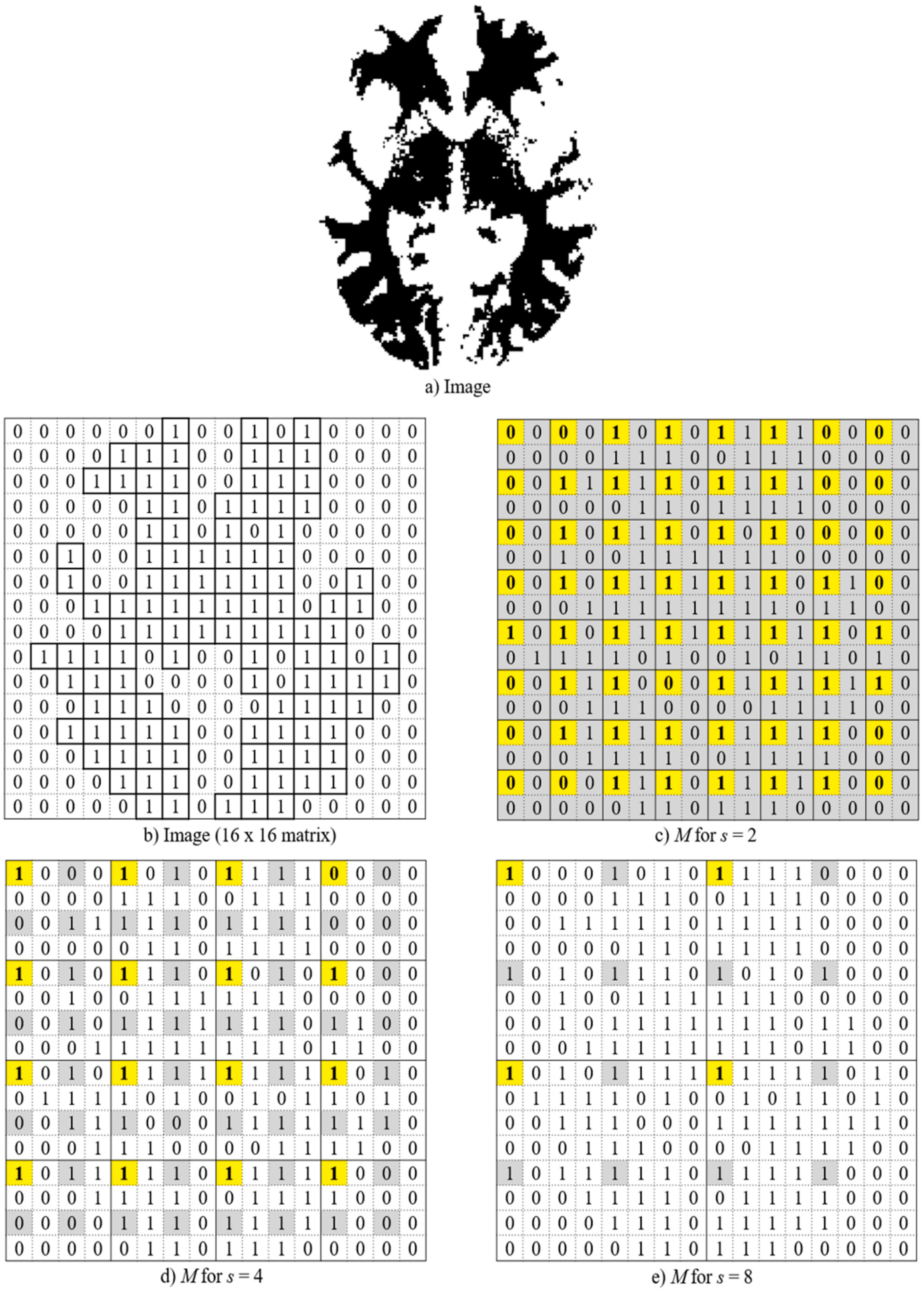

Fig. 1. Box occupancy computation in each step of the BC algorithm.

The main contributions of our proposal are: (1) We provide, to the best of our knowledge, the fastest GPU implementation of the BC algorithm in the literature. (2) Our fast BC algorithm can be run in any PC equipped with a GPU, which allows the researchers to process large datasets without the need for a supercomputer. (3) Our GPU algorithm was carefully designed to minimize GPU-CPU data transfers, computing the BC through very simple and efficient bit operations (avoiding divergent branches), and providing a launch configuration based on the optimal GPU multiprocessor occupancy; as a result, our GPU algorithm performs much faster than the single and multi-core CPU versions. (4) We provide particularized CUDA implementations of the BC algorithm for processing data in 2D, 3D and 4D, which covers a wide spectrum of possible datasets. (5) Finally, we also provide the source code of our GPU algorithms.

In the rest of the paper, we first describe the BC algorithm and explain how we have implemented it in GPU with CUDA. Next, we show the performance analysis of our CUDA algorithms for two different hardware platforms. Then we also compare the performance of our GPU BC algorithm against previous GPU implementations. Finally, we tested our BC algorithm in a biomedical context comparing the 4DFD of reconstructed EEG sources from a patient with schizophrenia and a healthy control while they were resting with open eyes.

2. Computational methods and theory

2.1. The BC algorithm

The FD of an ideal fractal set $F \in \mathbb{R}^d$ is computed by using the BC algorithm as follows [1,2]:

$$\mathrm{FD}(F) = \frac{\log(n(s))}{\log\left(\frac{1}{s}\right)} \tag{1}$$

where $n(s)$ is the number of boxes of scale s^d that are required to completely cover the fractal set F. If set F is not an ideal fractal, for example an image in 2D or a voxelized volume in 3D, then the FD is estimated as the linear regression of $\log(n(s))$ against $\log\left(\frac{1}{s}\right)$ for several values of s.

Biswas et al. [19] proposed a theoretical parallel version of the BC algorithm based on a Single Instruction-Multiple Data (SIMD) model with shared memory. This implementation selected a set of grid sizes s^d with s as a power of two. These grid sizes are a subset of all possible grid sizes, and therefore the FD value obtained by linear regression is almost identical to that obtained with all possible grid sizes. This selection of grid sizes allows the BC algorithm to perform efficiently in parallel, avoiding the need to traverse the boxes of set F again and again for each value of s. However, only the algorithmic complexity of the algorithm was shown in Biswas et al.'s study, without providing any implementation nor experimental result. In the present study we used the grid sizes selection and how the combination of results in each grid is performed in Biswas et al.'s BC algorithm as the base of our parallel implementation.

In order to compute the BC of set F, we use as data structure a binary matrix M of d dimensions. This matrix can be thought of as a voxelization or discretization of F, where a position of M has a value of 1 if this position contains some portion of F, or 0 otherwise. Fig. 1 shows an example in 2D ($d = 2$). In this example, F, the binary image in Fig. 1.a (a binarized slice of a magnetic resonance image), is represented by means of the matrix M of 16×16, as shown in Fig. 1.b.

According to Biswas et al.'s algorithm, the matrix M, of size m^d, is divided into non-overlapping grids of size s^d. This value s is the grid size, and ranges from 2 to $\frac{m}{2}$ in values which are to the power of two. For each grid i of size s^d, $n(i)$, the occupancy of the grid i, is computed as:

$$n(i) = i_1 \text{ OR } i_2 \text{ OR } \dots \text{ OR } i_s^d \tag{2}$$

where i_j are the matrix positions of the grid i of size s^d. This means that $n(i)$ has a value of 1 if any of its positions has a value of 1 (contains some portion of F), or 0 otherwise (no portion of F is contained in the grid i). Then the box-counting for size s is calculated as:

$$n(s) = \sum_i n(i), s \text{ ranging } \textit{from } 2 \text{ to } \frac{m}{2} \tag{3}$$

Finally, the FD of F is computed as the slope of the linear fitting of all the points $\left(log\left(\frac{1}{s}\right), \log(n(s))\right)$.

For the example in Fig. 1 ($d = 2$) the matrix M is initialized with the values of the binarized image (see Fig. 1.b). For $s = 2$, the occupancy values for each 2×2 grid i are obtained, the corresponding $n(i)$ values are calculated according to Eq. (2), and $n(s)$ is obtained through Eq. (3). Then the values of $n(i)$ are stored in the (0, 0) positions of each 2×2 grid i (see yellow cells in Fig. 1.c). For $s = 4$ (the next power of two value for s) the occupancy of each 4×4 grid can now be obtained without needing to compare the 16 values that each grid contains, but only comparing the previously-stored values in the four 2×2 grids (gray cells in Fig. 1.d) that each 4×4 grid contains. And again, these values are stored in the (0, 0) positions of each 4×4 grid i (see yellow cells in Fig. 1.d). These $n(i)$ values are finally added in order to obtain $n(s)$ for $s = 4$. Similarly, for $s = 8$ the occupancy of each 8×8 grid can also be obtained without needing to compare the 64 values it contains, but comparing only the four values previously stored for the corresponding four 4×4 grids (see Fig. 1.e). This process is repeated until s reaches $\frac{m}{2}$, obtaining in this way the set of values $n(s)$ which are the box-counting values needed to compute the FD of F.

A sequential C implementation of the BC algorithm for $d = 2$ is shown in Listing 1. Array n stores the box-counting for the matrix M of size $m \times m$, i.e. the $n(s)$ values for s ranging from 2 to $m/2$.

2.2. Parallel BC algorithm

As can be noted, the contribution of each grid of size s^d in Equation 2 (value of $M[i,j]$ in lines 9 and 10 in Listing 1) can be performed in parallel because this computation does not depend on the values stored in the other grids. In this way, we used OpenMP directives [20] to develop the parallel BC algorithm for execution in multi-core CPU, as shown in Listing 2.

The OpenMP directive *parallel for* in line 9 is used to divide the loop iterations (lines 10 and 11 in Listing 2) between a set of spawned threads. Each thread calculates the box occupancy, $n(i)$, for the corresponding grid (line 12). Then the final value of $n(s)$ (variable *sum* in line 13) is computed through parallel reduction sums (clause "*reduction*(+: *sum*)" in line 9). In line 6, the while-loop cannot be parallelized because values in M in one iteration depend on values stored in M in the previous iteration.

2.3. BC algorithm on GPU for 2D data

In this study we used CUDA [18] to implement the GPU version of the BC algorithm. The data is processed in CUDA through *kernels*, which are called as many times as threads are needed to process the data in an SIMD way. Threads are hierarchically grouped into *blocks* and blocks into *grids*. Threads inside a block are executed in groups (*warps*) of 32. On the other side, a CUDA GPU is a set of *multiprocessors* (MP), each one containing a set of *scalar processors* (SP or core). Each core has its own local memory and registers. Each MP has its own memory (*shared memory*) and can also access the main memory of the GPU (*global memory*). The CUDA programming model and the GPU hardware match: a grid of blocks is assigned to a GPU device, each block is assigned to an MP and each thread is assigned to a core.

Listing 3 shows the C implementation of our CUDA BC algorithm in 2D ($d = 2$). First, the matrix M is transferred from the RAM of the CPU to the global memory of the GPU. This is a main issue in the design of any

Listing 1
C implementation of the sequential version of the BC algorithm in 2D ($d = 2$).

```c
void seqBC(unsigned char M[m,m], const int m, unsigned long n[]) {
  unsigned int s = 2; // grid size is s x s
  unsigned int size = m;
  unsigned char ni = 0; // index for array n

  while (size > 2) {
    for(unsigned long i = 0; i < (m - 1); i=i+s) {
      for(unsigned long j = 0; j < (m - 1); j=j+s) {
        M[i,j] = M[i,j] || M[i,j + s/2] || M[i + s/2,j] || M[i + s/2,j + s/2];
        n[ni] += M[i,j]; // add the contribution of box (i,j) of size s
      }
    }
    ni++;
    s = s * 2;
    size = size / 2;
  }
}
```

Listing 2
C code with the parallel OpenMP version of the BC algorithm in 2D ($d = 2$).

```c
void parBC(unsigned char M[m,m], const int m, unsigned long n[]) {
  unsigned int s = 2; // grid size is s x s
  unsigned int size = m;
  unsigned char ni = 0; // index for array n

  while (size > 2) {
    unsigned long sum = 0; // needed for parallel reduction sums

    #pragma omp parallel for reduction(+:sum) // parallel computation of n[ni]
    for(unsigned long i = 0; i < (m - 1); i=i+s) {
      for(unsigned long j = 0; j < (m - 1); j=j+s) {
        M[i,j] = M[i,j] || M[i,j + s/2] || M[i + s/2,j] || M[i + s/2,j + s/2];
        sum += M[i,j]; // sum is computed by using reduction sums
      }
    }
    n[ni++] = sum;
    s = s * 2;
    size = size / 2;
  }
}
```

GPU algorithm, since data transfer between CPU and GPU slow down the overall performance of the program. Trying to optimize this step we used *pinned memory* in *cudaMallocHost*() calls in lines 3 and 6 in Listing 3. These calls avoid the cost of the intermediate transfer between pageable and pinned host arrays [21]. Then in line 23 of Listing 3 the CUDA kernel *BCKernel2D* is repeatedly called in order to compute the box-counting for each grid of size s. In each kernel call each thread processes a grid, so the number of blocks to launch is the number of grids in the matrix M divided by the number of threads per block (*TPB*). The TPB parameter has relevant implications in the overall performance of CUDA algorithms, so we explain in detail how this parameter was configured next in section 3.1. Once the kernel is launched, each thread computes the occupancy of the corresponding grid of size s and updates the value of n (s) in the GPU array *device_n*. Finally, in line 28 of Listing 3 the array which stores the box-counting of the matrix M is transferred from the global memory of the GPU (array *device_n*) to the RAM of the CPU (array n).

Listing 4 shows the CUDA code of the kernel for computing the occupancy of a grid of size $s \times s$ (*BCKernel2D* in Listing 3). Each thread computes and stores the occupancy value of the corresponding grid (line 17 in Listing 4) following Eq. (2), and adds this value to the position of the array n storing $n(s)$, the box-counting for size s (line 20 in Listing 4). Initially, the matrix M contains the occupancy values (0 or 1) of all grids of size s / 2. After the execution of all threads for size s, the matrix M contains the updated occupancy values for grids of size s. The computation of $n(s)$ is performed through atomic sums. This kind of operation was highly optimized from the NVIDIA Kepler architecture, so implementing $n(s)$ with a classical reduction sums approach did not provide better performance [22].

2.4. BC algorithm on GPU for 3D and 4D data

The BC algorithm for matrices of three dimensions is similar to the 2D case but considering one additional dimension when constructing grids and boxes. This means that each grid of size $s \times s \times s$ needs to evaluate eight positions in order to compute its occupancy. In order to compute the BC of a 3D matrix using the sequential CPU version (Listing 1) or the parallel OpenMP algorithm (Listing 2), another for-loop is required to traverse the third dimension. Then the computation of the occupancy is performed similarly by checking the eight values of each 3D grid.

In the CUDA algorithm (Listing 3), the launch configuration of the kernel for 3D is performed with $\frac{m}{s}\cdot\frac{m}{s}\cdot\frac{m}{s}$ blocks of *TPB* threads. Listing 5 shows the implementation of the CUDA kernel for computing the box-

Listing 3
C code of the CUDA BC algorithm in 2D ($d = 2$).

```
void cudaBC(unsigned char I[m,m], const int m, unsigned long n[]) {
  // allocating CPU pinned memory
  cudaMallocHost((void**) &M, sizeof(unsigned char) * m * m);
  memcpy(M, I, sizeof(unsigned char) * m * m);
  const int nn = log2(m) - 1; // number of elements in array n
  cudaMallocHost((void**) &n, sizeof(unsigned long) * nn);

  // CPU-GPU data transfers
  unsigned char *device_M; // matrix M in GPU device
  unsigned long *device_n; // array n in GPU device
  cudaMalloc((void**) &device_M, m * m * sizeof(unsigned char));
  cudaMemcpy(device_M, M, m * m * sizeof(unsigned char), cudaMemcpyHostToDevice);
  cudaMalloc((void**) &device_n, nn * sizeof(unsigned long));
  cudaMemset(device_n, 0, nn * sizeof(unsigned int));

  const unsigned char bits_m = nn + 1; // 2^bits_m = m
  unsigned int s = 2;
  unsigned char bits_TPB = log2(TPB); // 2^bits_TPB = TPB
  unsigned int size = m;
  unsigned char ni = 0;
  while (size > 2) {
    // BCKernel call. Compute box-counting for grids of size s x s
    BCKernel2D<<<((m*m)/(s*s))/TPB, TPB>>>(device_M, m, bits_m, s/2, ni+1, bits_TPB, &device_n[ni++]);
    s = s * 2;
    size = size / 2;
  }
  // GPU-CPU data transfer of the box-counting
  cudaMemcpy(n, device_n, nn * sizeof(unsigned long), cudaMemcpyDeviceToHost);
}
```

Listing 4
CUDA code computing the occupancy and box-counting of grids of size $s \times s$. $2^{bits\_m} = m$, $2^{bits\_s} = s$, $sm = s/2$ and $2^{\text{bits\_TPB}} = TPB$.

```
__global__ void BCKernel2D(unsigned char* M,const int m,const unsigned char bits_m, const unsigned int sm,
                           const unsigned char bits_s, const unsigned char bits_TPB, unsigned long* n) {
  register unsigned int tid = threadIdx.x;
  register unsigned int idx = (blockIdx.x << bits_TPB) + tid; // 2 ^ bits_TPB = TPB

  // identifies grid index (i, j) from block and thread values
  register unsigned int i = idx >> (bits_m - bits_s); // i index: idx / (m/s)
  register unsigned int j = idx & ((m >> bits_s) - 1); // j index: idx mod (m/s)

  // global location of position (0, 0) of the grid (i, j) is gi + gj
  const register unsigned int gi = (i << bits_s) << bits_m;
  const register unsigned int gism = ((i << bits_s) + sm) << bits_m;
  const register unsigned int gj = j << bits_s;
  const register unsigned int gjsm = (j << bits_s) + sm;

  // compute and store the occupancy value of the grid (i, j)
  M[gi + gj] = M[gi + gj] || M[gi + gjsm] || M[gism + gj] || M[gism + gjsm];

  // add the occupancy for grid (i, j) to box-counting n
  atomicAdd(n, M[gi + gj]);
}
```

counting of 3D grids of size $s \times s \times s$. In this kernel, the index k (line 8 in Listing 5) is added in order to correctly access the eight values of each grid of size $s \times s \times s$. The variable n stores the box-counting with the sum of the occupancies of all 3D grids in the matrix M.

Similar modifications were performed in all BC algorithms in order to process 4D matrices. Each 4D grid has a size of $s \times s \times s \times s$, so an additional for-loop was included to process the fourth dimension in Listing 1 and Listing 2. In this case, the occupancy of each grid is computed by checking the values of sixteen positions. The launch configuration of the CUDA kernel in 4D consisted of $\frac{m}{s}\frac{m}{s}\frac{m}{s}\frac{m}{s}$ blocks of TPB threads. Finally, in Listing 6 we show the CUDA kernel for processing 4D grids. The index l (line 9 in Listing 6) is added in order to correctly manage the fourth dimension of each grid.

3. Performance analysis

In this section we assess the performance of the CUDA BC algorithms (2D, 3D and 4D) by comparing them with the corresponding CPU versions (sequential and parallel OpenMP). We have also compared our CUDA BC algorithm with a previous CUDA implementation.

Two different platforms (a PC and a Server) were used to test the performance of our BC algorithms. Table 1 shows the hardware

Listing 5
CUDA kernel for computing the occupancy and box-counting of 3D grids of size $s \times s \times s$.

```
__global__ void BCKernel3D(unsigned char* M,const int m,const unsigned char bits_m, const unsigned int sm,
                            const unsigned char bits_s, const unsigned char bits_TPB, unsigned long* n) {
  register unsigned int tid = threadIdx.x;
  register unsigned int idx = (blockIdx.x << bits_TPB) + tid; // 2 ^ bits_TPB = TPB

  // identifies grid index (i, j, k) from block and thread values
  register unsigned int mdivs = m >> bits_s;
  register unsigned int k = idx >> ((bits_m - bits_s) + (bits_m - bits_s)); // k index: idx/((m/s)*(m/s))
  register unsigned int offset = (idx & ((mdivs << (bits_m - bits_s)) - 1)); // idx mod ((m/s)*(m/s)),
                                                                              // offset inside k slice
  register unsigned int i = offset >> (bits_m - bits_s); // i index: offset / (m/s)
  register unsigned int j = offset & (mdivs - 1); // j index: offset mod (m/s)

  // global location of position (0, 0, 0) of the grid (i, j, k) is gk + gi + gj
  const register unsigned int gi = (i << bits_s) << bits_m;
  const register unsigned int gj = j << bits_s;
  const register unsigned int gk = ((k << bits_s) << bits_m) << bits_m;
  const register unsigned int gism = ((i << bits_s) + sm) << bits_m;
  const register unsigned int gjsm = (j << bits_s) +sm;
  const register unsigned int gksm = (((k << bits_s) + sm) << bits_m) << bits_m;

  // compute and store the occupancy value of the grid (i, j, k)
  M[gk+gi+gj] = M[gk+gi+gj]       || M[gk + gi + gjsm]   || M[gk + gism + gj]   || M[gk + gism + gjsm] ||
                M[gksm + gi + gj] || M[gksm + gi + gjsm] || M[gksm + gism + gj] || M[gksm + gism + gjsm];

  // add the occupancy for grid (i, j, k) to box-counting n
  atomicAdd(n, M[gk+gi+gj]);
}
```

Listing 6
CUDA kernel for computing the occupancy and box-counting of 4D grids.

```
__global__ void BCKernel4D(unsigned char* M,const int m,const unsigned char bits_m, const unsigned int sm,
                            const unsigned char bits_s, const unsigned char bits_TPB, unsigned long* n) {
  register unsigned int tid = threadIdx.x;
  register unsigned int idx = (blockIdx.x << bits_TPB) + tid; // 2 ^ bits_TPB = TPB

  // identifies grid index (i, j, k, l) from block and thread values
  register unsigned int mdivs = m >> bits_s;
  // l index: idx/((m/s)*(m/s)*(m/s))
  register unsigned int l = idx >> ((bits_m - bits_s) + (bits_m - bits_s) + (bits_m - bits_s));
  // idx mod ((m/s)*(m/s)*(m/s)): offset inside 3D matrix l
  register unsigned int offsetl = (idx & ((mdivs << ((bits_m - bits_s) + (bits_m - bits_s))) - 1));
  // k index: offsetl/((m/s)*(m/s))
  register unsigned int k = offsetl >> ((bits_m - bits_s) + (bits_m - bits_s));
  // offsetl mod ((m/s)*(m/s)): offset inside slice k
  register unsigned int offset = (offsetl & ((mdivs << (bits_M - bits_s)) - 1));
  register unsigned int i = offset >> (bits_m - bits_s); // i index: offset / (m/s)
  register unsigned int j = offset & (mdivs - 1); // j index: offset mod (m/s)

  // global location of position (0, 0, 0, 0) of the grid (i, j, k, l) is gl + gk + gi + gj
  const register unsigned int gi = (i << bits_s) << bits_m;
  const register unsigned int gj = j << bits_s;
  const register unsigned int gk = ((k << bits_s) << bits_m) << bits_m;
  const register unsigned int gl = (((l << bits_s) << bits_m) << bits_m) << bits_m;
  const register unsigned int gism = ((i << bits_s) + sm) << bits_m;
  const register unsigned int gjsm = (j << bits_s) +sm;
  const register unsigned int gksm = (((k << bits_s) + sm) << bits_m) << bits_m;
  const register unsigned int glsm = ((((l << bits_s) + sm) << bits_m) << bits_m) << bits_m;

  // compute and store the occupancy value of the grid (i, j, k, l)
  M[gl+gk+gi+gj] = M[gl+gk+gi+gj] || M[gl+gk+gi+gjsm]   || M[gl+gk+gism+gj]     || M[gl+gk+gism+gjsm]   ||
               M[gl+gksm+gi+gj]   || M[gl+gksm+gi+gjsm] || M[gl+gksm+gism+gj]   || M[gl+gksm+gism+gjsm] ||
               M[glsm+gk+gi+gj]   || M[glsm+gk+gi+gjsm] || M[glsm+gk+gism+gj]   || M[glsm+gk+gism+gjsm] ||
               M[glsm+gksm+gi+gj] || M[glsm+gksm+gi+gjsm]|| M[glsm+gksm+gism+gj]|| M[glsm+gksm+gism+gjsm];

  // add the occupancy for grid (i, j, k, l) to box-counting n
  atomicAdd(n, M[gl+gk+gi+gj]);
}
```

Table 1
Hardware capabilities of the platforms used to test the algorithms.

	Platform	
	PC	Server
Operating System	Windows 10 × 64	Debian Linux 5.10 × 86_64
CPU		
Model	Intel Core i7–11800 H @ 2.30 GHZ	2 x Intel Xeon CPU Silver 4210 @ 2.20 GHz
Cores – Threads	8 – 16	20 – 40
RAM	32 GB	96 GB
Power consumption	45 W	85 W
GPU		
Model	NVIDIA GeForce RTX 3060	NVIDIA GeForce RTX 3090
Computing Capability	8.6	8.0
CUDA SDK	11.0	11.0
Arqchitecture	Ampere	Ampere
# MPs	30	82
# SPs	3584	10,496
Warp Size	32	32
Maximum Threads per Block	1024	1024
Global Memory Size	6 GB	24 GB
Shared Memory per Block	48KB	48KB
Registers per Block	64 K	64 K
L2 Cache Size	3MB	6MB
Error Correcting Codes	Disabled	Disabled
Power consumption	170 W	350 W

configuration of each platform.

In order to adequately evaluate the performance and the scalability of our CUDA BC algorithms, the execution of each algorithm was tested on a set of four matrices for each dimension. The sizes of these matrices were: 4096^2, 8192^2, 16384^2 and 32768^2 (2D); 128^3, 256^3, 512^3 and 1024^3 (3D); and 32^4, 64^4, 128^4 and 256^4 (4D). The values of all of these matrices were randomly generated using the same seed in order to guarantee that all of the algorithms tested processed the same matrix.

3.1. *Configuration of threads per block (TPB) parameter*

The number of threads per block (TPB) used when launching the CUDA kernels has a direct impact on performance, since TPB determines the percentage of use of each GPU multiprocessor [23], known as *MP occupancy*. In order to adequately set the TPB parameter, we analyzed the actual MP occupancy achieved by our CUDA kernels for TPB values ranging from 32 to 1024 by using the CUDA Compute Visual Profiler tool [24]. Fig. 2 shows the results of this analysis for the 4D case (CUDA kernel BCKernel4D). In this figure the kernel speedup of the algorithm indicates the improvement of CUDA algorithm regarding the CPU version without taking into account the data transfers between GPU and CPU. The theoretical MP occupancy was obtained as the ratio of active threads to the maximum number of threads supported on the MP (see Table 1).

Almost equal values of MP occupancy were obtained at 96 threads per block: 89.4% for RTX3060 (PC) and 89.5% for RTX 3090 (Server). Our CUDA kernels use a small amount of registers per thread and no shared memory storage is needed, and these two factors are key in obtaining high values of MP occupancy [23]. Finally, we selected the TPB values which achieved the best speedups: 96 (PC) and 128 (Server) in 2D; 96 (PC) and 512 (Server) in 3D; and 96 (PC) and 128 (Server) in 4D.

3.2. *Performance of CUDA BC algorithms*

We tested the performance of our CUDA BC algorithms (*cudaBC* in Listing 3 with kernels for 2D in Listing 4, 3D in Listing 5 and 4D in Listing 6) by comparing them to the two CPU versions (sequential version *seqDBC* in Listing 1 and the parallel OpenMP version *parDBC* in Listing 2). All implementations were tested on the two hardware platforms (see Table 1) using four incremental-size matrices for each dimension. Table 2 shows the timing and speedups achieved.

The sequential algorithm (*seqBC*) performed better on the PC platform than on the Server because of the computational power of the individual cores in both CPUs. Nevertheless, the multi-threaded OpenMP CPU implementation executed much faster on the Server due to the difference in the number of cores between both CPUs. Top speedups of 5.83 × (2D), 5.58 × (3D) and 6.31 × (4D) were obtained when comparing the parallel CPU version against the sequential algorithm (seqBC/parBC in Table 2) on the PC platform; and 25.46 × (2D), 23.90 × (3D) and 21.34 × (4D) when comparing on the Server platform. These speedup results show that the OpenMP parallel algorithm

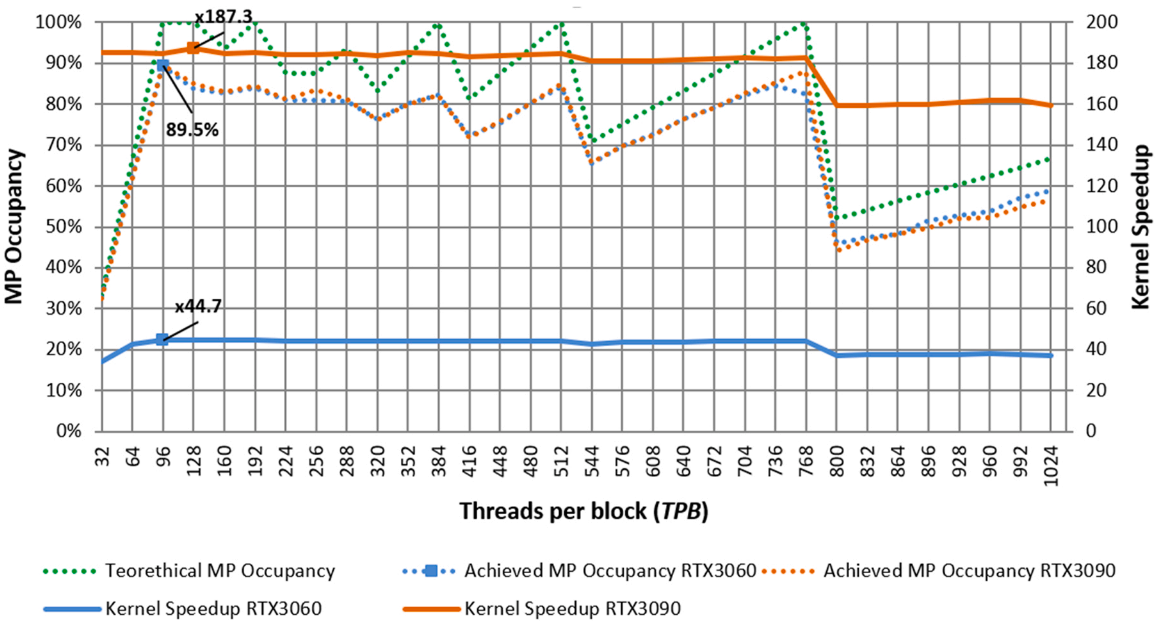

Fig. 2. MP occupancy and kernel speedup achieved by BCKernel4D for different values of TPB.

Table 2
Execution times and speedups of the BC algorithms. CUDA times include data transfers between CPU and GPU. Average values for ten executions. Time expressed in seconds. Bold values correspond to the highest speedups achieved on each platform.

Dim.	Size	seqBC (s)		parBC (s)		cudaBC (s)		seqBC/parBC		seqBC/cudaBC		parBC/cudaBC	
		PC	Server	PC	Server	PC	Server	PC	Server	PC	Server	PC	Server
2D	4096^2	0.1105	00.186	0.0205	0.0120	0.0026	0.0027	5.39 ×	15.46 ×	41.88 ×	67.64 ×	7.77 ×	**4.38 ×**
	8192^2	0.4081	00.740	0.0803	0.0419	0.0059	0.0106	5.08 ×	17.64 ×	68.66 ×	**69.68 ×**	13.51 ×	3.95 ×
	16384^2	1.6825	02.954	0.2887	0.1693	0.0195	0.0436	**5.83 ×**	17.44 ×	85.88 ×	67.68 ×	14.74 ×	3.88 ×
	32768^2	6.8252	11.901	1.1909	0.4674	0.0738	0.1765	5.73 ×	**25.46 ×**	**92.38 ×**	67.40 ×	**16.12 ×**	2.65 ×
3D	128^3	0.0147	00.024	0.0032	0.0021	0.0013	0.0005	4.60 ×	11.08 ×	11.24 ×	44.47 ×	2.44 ×	4.01 ×
	256^3	0.1114	00.177	0.0260	0.0160	0.0038	0.0030	4.27 ×	11.07 ×	29.31 ×	**57.27 ×**	**6.86 ×**	**5.17 ×**
	512^3	0.9566	01.415	0.1880	0.0683	0.0318	0.0268	5.09 ×	20.70 ×	30.02 ×	52.68 ×	5.90 ×	2.54 ×
	1024^3	7.4656	11.323	1.3379	0.4738	0.2397	0.2526	**5.58 ×**	**23.90 ×**	**31.14 ×**	44.82 ×	5.58 ×	1.88 ×
4D	32^4	00.010	00.015	0.0027	0.0009	0.0008	0.0002	3.79 ×	17.20 ×	12.22 ×	53.04 ×	3.22 ×	3.08 ×
	64^4	00.163	00.220	0.0259	0.0117	0.0034	0.0029	**6.31 ×**	18.72 ×	**47.22 ×**	**75.73 ×**	**7.49 ×**	**4.04 ×**
	128^4	02.397	03.468	0.4345	0.1828	0.0623	0.0569	5.52 ×	18.97 ×	38.45 ×	60.91 ×	6.97 ×	3.21 ×
	256^4	37.769	56.339	6.5587	2.6402	1.0003	0.8878	5.76 ×	**21.34 ×**	37.76 ×	63.46 ×	6.56 ×	2.97 ×

achieved very high utilization of the multi-core CPU for the PC platform (74% on average), and a value even greater than full utilization for the Server platform (118% on average) due to hyper-threading technology.

Very high top speedups were achieved on both platforms when comparing the performance of the CUDA BC algorithm against the sequential CPU version (seqBC/cudaBC in Table 2): 92.38 × (2D), 31.14 × (3D) and 47.22 × (4D) on the PC platform, and 69.68 × (2D), 57.27 × (3D) and 75.73 × (4D) on the Server platform. Compared to the parallel OpenMP implementation (parBC/cudaBC in Table 2), the CUDA algorithm achieved speedups of 16.12 × (2D), 6.86 × (3D) and 7.49 × (4D) on the PC platform, and 4.38 × (2D), 5.17 × (3D) and 4.04 × (4D) on the Server platform. These results indicate that the CUDA BC algorithm performs very efficiently and scales well when the dimension of the data increases.

Data transfers between CPU and GPU have an important impact on the overall performance of our CUDA algorithm. These data transfers need similar times in both platforms, and this is the reason why the overall computation times of the CUDA algorithms do not reflect the actual difference in computation power between the GPUs of both platforms. Nevertheless, that difference in GPU performance is clearly shown in kernel speedups, where timing does not consider data transfers (see Fig. 3). Top kernel speedups of 465.74 × (2D), 173.25 × (3D) and 257.93 × (4D) were obtained for the Server platform, while top kernel speedups of 248.82 × (2D), 38.08 × (3D) and 54.38 × (4D) were achieved on the PC platform (points with black border in Fig. 3).

The speedups of the CUDA BC algorithms (2D, 3D and 4D) achieved on the two platforms are shown graphically in Fig. 3. For the 2D case, when the size of the matrix increases, the speedup trends also grow. This fact implies that the CUDA algorithm scales very well. This situation also occurs in 3D and 4D on the PC platform. On the server platform the speedup trend plateaus, but still at very high values.

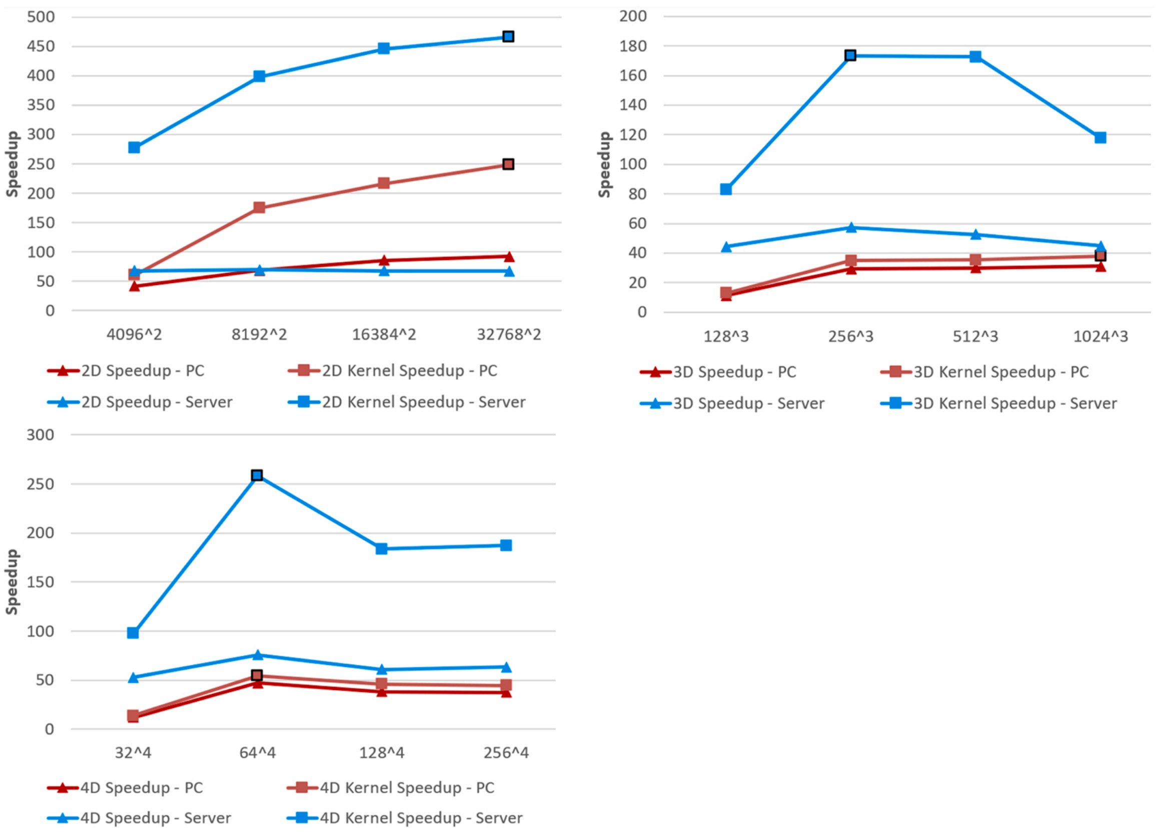

Fig. 3. 2D, 3D and 4D Speedups (seqBC / cudaBC) and kernel speedups: (seqBC / cudaBC, without CPU-GPU transfers).

3.3. Comparison with previous work

As shown in Section 1, there are not many studies about optimizing the computational performance of the BC algorithm. Hou et al.'s algorithm [15] and its later optimization by Nikolaides et al. [16] were presented to be executed only sequentially on CPU. Jiménez et al. [17] presented a fast GPU implementation of Hou et al.'s algorithm based on CUDA, the only GPU BC algorithm to the best of our knowledge.

Table 3 shows the execution times and speedups of the comparisons between Jiménez et al.'s CUDA algorithm (code provided by authors) and our GPU BC algorithm. These comparisons were also performed on PC and Server platforms. Jiménez et al.'s code can only process 3D matrices, so we compared it to our *cudaBC* algorithm (Listing 3) with *BCKernel3D* (Listing 5). Our CUDA algorithm obtained speedups of up to 4.62 × (PC) and 4.79 × (Server) when considering data transfers (*cudaBC* and *cudaJim*), and speedups of up to 4.31 × (PC) and 7.33 × (Server) when only the time of the kernel execution was considered (*cudaBCk* and *cudaJimk*).

The high speedups achieved by our GPU algorithm are mainly due to the fact that Jiménez et al.'s GPU algorithm, like Hou et al.'s original BC algorithm, needs to create and sort a list with the binary representation of non-zero positions of the matrix. Although this list allows the algorithm to compute the occupancy of grids very efficiently, the computation time needed to create and sort it, even using highly optimized CUDA libraries such as Thurst [23], prevent Jiménez et al.'s GPU implementation from being competitive against our GPU algorithm which does not need any pre-processing step.

4. A case study: 4D FD analysis of the EEG signal in schizophrenia

In order to test our fast BC algorithm in a real biomedical application, we performed a 4D FD analysis of the EEG signal in schizophrenia. Two subjects were included in this study, recruited at the University Hospital of San Agustín Linares (Jaén, Spain): one subject suffering from schizophrenia and one healthy control subject. Resting state EEG data was acquired in a three-minute session where participants sat in a laboratory room at the hospital. The cap consisted of 31 electrodes in the 10–20 system. Signals were recorded at a frequency of 500 Hz. Blinks and other artifacts were extracted using the infomax ICA algorithm in EEGLAB.

Source modeling was performed in order to localize 15,000 primary electromagnetic sources of scalp EEG activity at each sample time. After source modeling a binarization process was performed in order to identify the significant sources. The point clouds described by the 3D localizations associated with the significant sources at each time sample define the 4D matrices with the spatiotemporal representation of brain activation. Details of the whole process for obtaining the 4D matrices from the EEG data were provided previously in [7].

For the present study, we selected for analysis the twenty central seconds of the range of three minutes. Due to the size limitation of the global memory of the GPU, an individual 4D matrix of size 256^4 was created for each second of brain activation. This 4D matrix represented a down-sampling to 256 time-samples (fourth dimension) of the brain activation spatially located in 3D grids of size 256^3. Therefore, twenty 4D matrices were processed for each subject.

Table 4 shows the computation times for processing the whole experiment (40 4D matrices) by executing the three versions of our BC algorithm (*seqBC*, *parBC* and *cudaBC*) and also using a widely-referred MATLAB implementation [25] modified to be able to process 4D matrices. The experiment took less than three minutes for our CUDA algorithm, while more than three hours were required when executing on sequential CPU. The MATLAB algorithm required sixteen hours to complete the experiment.

The 4D FD values obtained for the twenty matrices of each subject are listed in Table 5 and graphically represented in the boxplot shown in Fig. 4. As can be seen, the 4D FD differentiates the brain activation in resting state between subjects very well. According to the t-student test, 4D FD is significantly lower for the schizophrenia patient ($t(18) = 10.12$; $p < 0.001$).

5. Conclusions

We have presented an efficient BC algorithm on GPU, valid for 2D, 3D and 4D data, which greatly outperforms the execution-time compared to the sequential and parallel OpenMP CPU versions. A set of matrices of incremental sizes have been used to test the performance of the algorithms on two different hardware platforms. Our GPU BC algorithm achieved speedups of up to 92.38 × with respect to the sequential CPU implementation, and speedups of up to 16.12 × compared to the parallel OpenMP CPU version.

Currently, to our knowledge the only GPU implementation of the BC algorithm is the one presented by Jiménez et al. [17]. Compared with this GPU implementation our CUDA algorithm achieved a speedup of up to 4.79 × in 3D.

We have also proved the usefulness of our GPU BC algorithm by applying it in a spatiotemporal 4D FD analysis of EEG data in schizophrenia. Our fast BC algorithm was able to reduce the computation time of 40 4D matrices, extracted from the EEG of two subjects, from sixteen hours (MATLAB) and three hours (sequential CPU) to less than three minutes with the GPU algorithm. This experiment also showed 4D FD as a promising measure for differentiating schizophrenia patients from healthy controls. These preliminary results need to be further validated with a larger representative sample.

The CUDA source code of our GPU BC algorithms is publicly available and can be downloaded from https://www.ugr.es/~demiras/fbc.

CRediT authorship contribution statement

J. Ruiz de Miras: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Writing – original draft, Funding acquisition. **M.Á. Posadas**: Conceptualization, Methodology, Software, Data curation, Writing – review & editing. **A.J. Ibáñez-Molina**: Methodology, Investigation, Resources, Data curation, Writing – review & editing. **M.F. Soriano**: Methodology, Investigation, Resources, Data curation, Writing – review & editing. **S. Iglesias-Parro**: Methodology, Investigation, Resources, Data curation, Writing – review & editing, Funding acquisition.

Table 3
Execution times and speedups comparing our CUDA BC algorithm (cudaBC) with Jiménez's CUDA implementation (cudaJim). Times for cudaBCk and cudaJimk do not include data transfers between CPU and GPU. Average values for ten executions. Time expressed in seconds.

Size	cudaBC (s)		cudaJim (s)		cudaBCk (s)		cudaJimk (s)		cudaJim/cudaBC		cudaJimk/cudaBCk	
	PC	Server	PC	Server	PC	Server	PC	Server	PC	Server	PC	Server
64^3	0.00077	0.00021	0.00144	0.00103	0.00073	0.00017	0.00122	0.00084	1.86 ×	**4.79**×	1.67 ×	4.93 ×
128^3	0.00131	0.00058	0.00397	0.00223	0.00119	0.00030	0.00300	0.00220	3.04 ×	3.81 ×	2.51 ×	**7.33** ×
256^3	0.00380	0.00312	0.01754	0.00882	0.00319	0.00115	0.01378	0.00651	**4.62**×	2.83 ×	**4.31**×	5.62 ×
512^3	0.03186	0.02524	0.12259	0.05495	0.02632	0.00823	0.10379	0.03751	3.85 ×	2.18×	3.94 ×	4.56 ×

Table 4
Execution times and speedups for computing the 4D FD values corresponding to 40 4D matrices of size 256^4 (20 4D matrices for the schizophrenia patient and 20 4D matrices for the healthy control). Time expressed in seconds. Computations performed in the Server platform.

Size	MATLAB (s)	seqBC (s)	parBC (s)	cudaBC (s)	MATLAB/ cudaBC	seqBC/ cudaBC	parBC/ cudaBC
40×256^4	57,772.25	8393.20	496.91	171.04	337.77 ×	49.07 ×	2.90 ×

Table 5
4D FD values obtained for each second of the EEG data analyzed. 20 4D FD values for the schizophrenia patient (SCZ) and 20 4D FD values for the healthy control subject (HC).

Subject	4D FD values										Avg.	Std.
HC	3.327	3.311	3.314	3.329	3.320	3.308	3.330	3.318	3.313	3.310	3.324	0.011
	3.316	3.330	3.344	3.330	3.349	3.318	3.321	3.333	3.345	3.329		
SCZ	3.270	3.269	3.270	3.286	3.259	3.193	3.234	3.238	3.275	3.275	3.259	0.026
	3.249	3.274	3.252	3.257	3.217	3.305	3.253	3.240	3.270	3.298		

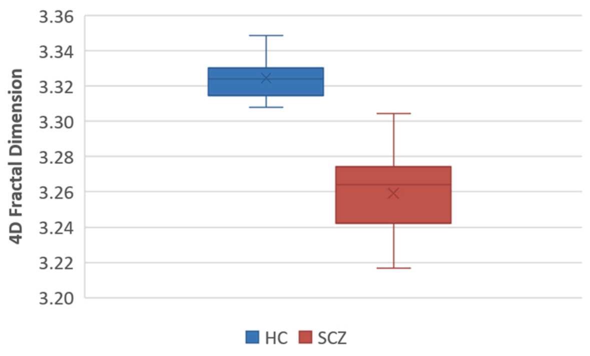

Fig. 4. Box plot showing differences in 4D FD values between the schizophrenia patient (SCZ) and the healthy control subject (HC). 20 4D FD values for each subject, one value for each second of the EEG data analyzed.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability

Data will be made available on request.

Acknowledgment

This work is part of the research project PID2019-105145RB-I00 supported by the Spanish Government (MCIN/AEI/10.13039/501100011033).

References

[1] D.A. Russell, J.D. Hanson, E. Ott, Dimension of strange attractors, Phys. Rev. Lett. 45 (1980) 1175–1178, https://doi.org/10.1103/PhysRevLett.45.1175.
[2] B.B. Mandelbrot, The Fractal Geometry of Nature, 1983. ⟨https://doi.org/10.1119/1.13295⟩.
[3] A.C. de Mattos, J.B. Florindo, R.L. Adam, I. Lorand-Metze, K. Metze, The fractal dimension suggests two chromatin configurations in small cell neuroendocrine lung cancer and is an independent unfavorable prognostic factor for overall survival, Microsc. Microanal. (2022) 1–5, https://doi.org/10.1017/S1431927622000113.
[4] R. Ternifi, Y. Wang, E.C. Polley, R.T. Fazzio, M. Fatemi, A. Alizad, Quantitative biomarkers for cancer detection using contrast-free ultrasound high-definition microvessel imaging: fractal dimension, Murray's deviation, bifurcation angle & spatial vascularity pattern, IEEE Trans. Med. Imaging 40 (2021) 3891–3900, https://doi.org/10.1109/TMI.2021.3101669.
[5] C.-W. Jao, C.I. Lau, L.-M. Lien, Y.-F. Tsai, K.-E. Chu, C.-Y. Hsiao, J.-H. Yeh, Y.-T. Wu, Using fractal dimension analysis with the Desikan–Killiany atlas to assess the effects of normal aging on subregional cortex alterations in adulthood, Brain Sci. 11 (2021), https://doi.org/10.3390/brainsci11010107.
[6] V. Meregalli, F. Alberti, C.R. Madan, P. Meneguzzo, A. Miola, N. Trevisan, F. Sambataro, A. Favaro, E. Collantoni, Cortical complexity estimation using fractal dimension: a systematic review of the literature on clinical and nonclinical samples, Eur. J. Neurosci. n/a (2022), https://doi.org/10.1111/ejn.15631.
[7] J. Ruiz de Miras, F. Soler, S. Iglesias-Parro, A.J. Ibáñez-Molina, A.G. Casali, S. Laureys, M. Massimini, F.J. Esteban, J. Navas, J.A. Langa, Fractal dimension analysis of states of consciousness and unconsciousness using transcranial magnetic stimulation, Comput. Methods Prog. Biomed. 175 (2019) 129–137, https://doi.org/10.1016/j.cmpb.2019.04.017.
[8] A.P.H. Don, J.F. Peters, S. Ramanna, A. Tozzi, Quaternionic views of rs-fMRI hierarchical brain activation regions. Discovery of multilevel brain activation region intensities in rs-fMRI video frames, Chaos Solitons Fractals 152 (2021), 111351, https://doi.org/10.1016/J.CHAOS.2021.111351.
[9] I.V. Grossu, I. Grossu, D. Felea, C. Besliu, A. Jipa, T. Esanu, C.C. Bordeianu, E. Stan, Hyper-fractal analysis: a visual tool for estimating the fractal dimension of 4D objects, Comput. Phys. Commun. 184 (2013) 1344–1345, https://doi.org/10.1016/j.cpc.2012.11.018.
[10] S. Liu, W. Bai, N. Zeng, S. Wang, A fast fractal based compression for MRI images, IEEE Access 7 (2019) 62412–62420, https://doi.org/10.1109/ACCESS.2019.2916934.
[11] D. Hong, Z. Pan, X. Wu, Improved differential box counting with multi-scale and multi-direction: a new palmprint recognition method, Optik 125 (2014) 4154–4160, https://doi.org/10.1016/J.IJLEO.2014.01.093.
[12] H. Wang, B. Zhang, W. Chen, Robust and real-time object recognition based on multiple fractal dimension, Multimed. Tools Appl. 80 (2021) 36585–36603, https://doi.org/10.1007/s11042-021-11447-1.
[13] O.J. Escalona, M. Mendoza, G. Villegas, C. Navarro, Real-time system for high-resolution ECG diagnosis based on 3D late potential fractal dimension estimation, Comput. Cardiol. (2011) 789–792.
[14] X. Chen, W. Chang, Z. Gao, Method and parallel architecture for extracting the image fractal dimension in real time, Proc. SPIE (1998), https://doi.org/10.1117/12.311096.
[15] X.-J. Hou, R. Gilmore, G.B. Mindlin, H.G. Solari, An efficient algorithm for fast box counting, Phys. Lett. A 151 (1990) 43–46, https://doi.org/10.1016/0375-9601(90)90844-E.
[16] J. Nikolaides, E. Aifantis, Z-box merging: ultra-fast computation of fractal dimension and lacunarity, in: Proceedings of the IEEE 30th International Symposium on Computer-Based Medical Systems, 2017, pp. 312–317. ⟨https://doi.org/10.1109/CBMS.2017.121⟩.
[17] J. Jiménez, J. Ruiz de Miras, Fast box-counting algorithm on GPU, Comput. Methods Prog. Biomed. 108 (2012), https://doi.org/10.1016/j.cmpb.2012.07.005.
[18] NVIDIA Corporation, Cuda C++ Programming Guide v11.6, 2022. ⟨https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf⟩.
[19] M.K. Biswas, T. Ghose, S. Guha, P.K. Biswas, Fractal dimension estimation for texture images: a parallel approach, Pattern Recognit. Lett. 19 (1998) 309–313, https://doi.org/10.1016/S0167-8655(98)00002-6.
[20] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, 2007. ⟨https://ieeexplore.ieee.org/servlet/opac?bknumber=6267237⟩.
[21] N. Wilt, The CUDA Handbook, Addison-Wesley, 2013.
[22] J. Ruiz de Miras, M. Salazar, GPU inclusion test for triangular meshes, J. Parallel Distrib. Comput. 120 (2018) 170–181, https://doi.org/10.1016/j.jpdc.2018.06.003.
[23] D.B. Kirk. Programming Massively Parallel Processors: A Hands-on Approach, third ed., Morgan Kaufmann Publishers, 2016 https://doi.org/10.1016/B978-0-12-415992-1.00022-5.
[24] NVIDIA, Compute Visual Profiler, 2019. ⟨http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf⟩.
[25] F. Moisy, Boxcount, MATLAB Cent., 2008. ⟨https://www.mathworks.com/matlabcentral/fileexchange/13063-boxcount⟩.

J. Ruiz de Miras is a senior lecturer at the Software Engineering Department of the University of Granada (Spain). Dr. Ruiz de Miras received an MS degree in Computer Science in 1995 and a PhD in Computer Science in 2001. He was visiting Dr. Sepulcre's Brain Connectomics Lab in Harvard University (Boston - USA) for three months in 2015, and Dr. Angela Comanducci's Multimodal Neurophysiology Laboratory for Rehabilitation at Fondazione Don Carlo Gnocchi (Milan - Italy) for three months in 2022. His current main research interests are in Fractal Analysis, Optimization of geometric processing algorithms using GPU and multi-core CPU architectures, and Computing applied to medical image analysis and biomedicine.

M.A. Posadas is a postgraduate student collaborating at Software Engineering Department of University of Granada (Spain). He received a MS degree in Computer Science in 2021. His main research lines are in GPU programming and software engineering.

A.J. Ibáñez Molina started to research in the group of Language and Memory in the University of Granada (Spain). He also received a grant from the Spanish government (FPU) that allowed him to obtain a European PhD in 2009. In that period he researched 4 months in the University of Sussex (UK) as a Marie Curie fellow, and at the Radboud University in Nijmegen (The Netherlands). He also researched as a posdoc fellow in the Basque Center on Cognition Brain and Language where he collaborated with researchers from different countries and had the opportunity to learn advanced methods in time series analyses. After that, in the University of Jaén, Dr. Ibáñez met Professor Iglesias-Parro and in collaboration with him, he became interested in non-linear analyses applied to EEGs and whole brain neurocomputational models based on coupled differential equations. They developed these type of methods to investigate conscious states in healthy and patients with different disorders.

M.F. Soriano have worked as a Clinical Psychologist in a Day Hospital for severe mental disorders since 2005. Her PhD research was focused on cognitive dysfunctions in schizophrenia. The main interest of her research work has been the understanding of mental and neurophysiological undepinnings of the most relevant severe disorders. Dr. Sorianós research work has included patients with schizophrenia, but also with bipolar disorder and autism spectrum disorders, with publications in these areas. To this end, Dr. Sorianós have tried to integrate the concepts and methodology of Experimental Psychology and Neuroscience (acquired during my doctoral training) with her clinical training and experience.

S. Iglesias-Parro is a professor (tenured) in Psychometrics at the University of Jaén (Spain), and external postdoc lecturer at the Universidad Nacional de Educación a Distancia, Madrid, (Spain). Dr. Iglesias obtained a MS degree in Psychology from the University of Granada in 1993. Dr. Iglesias also completed the Doctoral Thesis in the University of Granada. Invited to research in the US, University of Duke, 2006, under the supervision of Prof. John Payne and at UNED (Madrid, Spain), 2012, under the supervision of Prof. Francisco Morales. Dr. Iglesias participated in numerous research projects and published more than 30 articles in prestigious journals, mainly related to two lines of research. A first line of research revolves around cognition in decision making. Dr. Iglesias studied the role of different variables of the automatic - controlled spectrum in decision making: memory, cognitive effort, prejudice, inhibition, among others. Deepening the issue of automaticity and cognitive control, Dr. Iglesias have been approaching his second line of research: consciousness. As a result of this interest Dr. Iglesias published several works in JCR indexed journals and participated in various research projects funded in competitive calls.