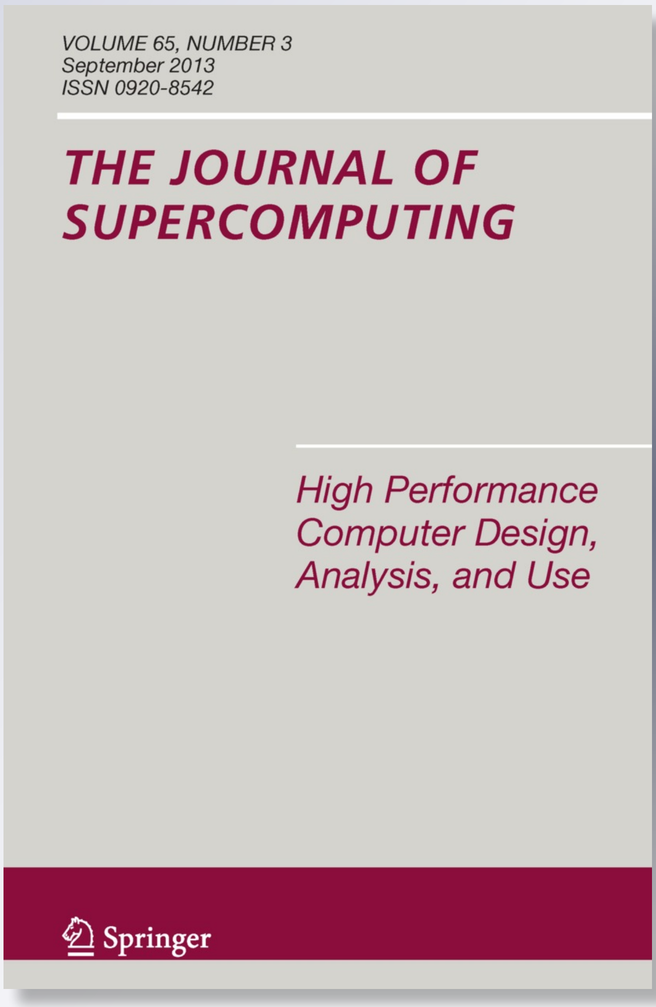
Box-counting algorithm on GPU and multi-core CPU: an OpenCL cross-platform study

Jesús Jiménez & Juan Ruiz de Miras

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542
Volume 65
Number 3

J Supercomput (2013) 65:1327-1352
DOI 10.1007/s11227-013-0885-z


VOLUME 65, NUMBER 3
September 2013
ISSN 0920-8542
THE JOURNAL OF SUPERCOMPUTING
High Performance Computer Design, Analysis, and Use
Springer

Springer

J Supercomput (2013) 65:1327–1352
DOI 10.1007/s11227-013-0885-z

Box-counting algorithm on GPU and multi-core CPU: an OpenCL cross-platform study

Jesús Jiménez · Juan Ruiz de Miras

Published online: 1 February 2013

Abstract In this paper, we present the analysis and development of a cross-platform OpenCL implementation of the box-counting algorithm, which is one of the most widely-used methods for estimating the Fractal Dimension. The Fractal Dimension is a relevant image analysis method used in several disciplines, but computing it is in general a time consuming process, especially when working with 3D images. Unlike parallel programming models that strictly depend on the hardware type and manufacturer, like CUDA, OpenCL allows us to provide an implementation suitable for execution on both GPUs and multi-core CPUs, whatever the hardware manufacturer. Sorting is a key part of the fast box-counting algorithm and the final speedup is highly conditioned by the efficiency of the sorting algorithm used. Our study reveals that current OpenCL implementations of sorting algorithms are clearly slower when compared with both CUDA for GPU and specific multi-core CPU implementations. Our OpenCL algorithm has been specifically optimized according the type of the target device and the results show an average speedup of up to 7.46× and 4×, when executed on the GPU and the multi-core CPU respectively, both compared with the single-threaded (sequential) CPU implementation.

Keywords Box-counting · OpenCL · GPU · CPU · Fractal dimension

1 Introduction

The fractal dimension (FD) is a quantitative parameter that characterizes the morphometric variability of a complex object. In recent years, several studies have showed the utility of the FD as a good descriptor of structures in a wide range of scientific

J. Jiménez · J. Ruiz de Miras (✉)
Department of Computer Science, University of Jaén, 23071 Jaén, Spain
e-mail: demiras@ujaen.es

fields, such as the biomedical field [1–3], materials analysis [4–6], environmental science [7], and computer graphics [8], among others. The FD is usually estimated by performing the box-counting calculation [9]. In general terms, the box-counting algorithm consists of counting how many square boxes, for different sizes, are needed to completely cover a model represented in a 2D image or a 3D volume.

Algorithms for image or volume analysis usually imply an extensive computing time, or often need an analysis of many datasets by applying repetitive processes. Thus, implementations of parallel and efficient algorithms in this field are highly welcome. Computing the box-counting for large models is not an exception: it is a time consuming process, especially when working with 3D models, and it is usually used on studies with hundreds of images. If the box-counting algorithm is optimized, making it as efficient and fast as possible, a performance improvement while calculating the FD will be achieved. As an example, in [10] a program for calculating the 3D FD of magnetic resonance images (MRI) is presented. Due to the increasing amount and size of the data, they exposed the need to optimize this software in terms of execution time by accelerating the 3D box-counting algorithm. In addition, it would be very important that this class of algorithms could be executed on different platforms, independently of the device type (GPU, CPU) or the hardware vendor (NVIDIA, AMD, Intel, . . .).

A simple, widely used and very efficient method of performing the box-counting calculation is proposed by Hou et al. [11]. This algorithm is an optimization of the one proposed by Liebotich et al. [12]. Hou's algorithm is the basic method in many studies, i.e. [5, 13, 14]. However this algorithm, although efficient, only uses one thread in its execution.

Nowadays, parallel computers are not the expensive and elitist devices they were years ago, since multi-core hardware is present in almost all PCs. Basically, there are two main multi-core approaches: to integrate a few cores into a single microprocessor (multi-core CPUs), or to integrate a large number of cores, exemplified by the current Graphical Processing Units (GPUs) [15]. We can exploit all these multi-core hardware by launching more than one execution thread.

It is clear that we can take advantage of multi-core CPUs by splitting the execution of a single-threaded (sequential) process in some parts that could be independently executed in each processing core. However, using many-core GPUs for parallelizing a general problem is still a difficult and low-level task. The use of the GPU for general purposes (GPGPU) has grown exponentially, mainly due to the appearance of new architectures and programming models such as NVIDIA CUDA [16] or Khronos OpenCL (Khronos Open Computing Language) [17]. These programming models use simple extensions of well-known languages like C, and they do not require any previous knowledge neither on GPU programming nor on the graphics visualization pipeline. In fact, we presented in [18] a CUDA implementation of Hou's box-counting algorithm for execution on NVIDIA's GPU devices, achieving great results in terms of accuracy and running time.

The OpenCL programming model has a main advantage over CUDA: the cross-platform character. Thus, with OpenCL we can code multi-threaded applications that can be executed on GPUs and also on multi-core CPUs, whatever the hardware manufacturer. In general terms, its programming model consists of a host (the main device) connected to and controlling one or more OpenCL devices. Thus, main sequential code runs on the host, and it submits work to the available OpenCL devices. The basic unit of work for an OpenCL device is called a "*work item*", which executes the code located in a function called kernel, which is invoked in as many work items as are necessary to process the dataset. Work items are grouped into "*work groups*" that could be mono-, bi- or three-dimensional; and work groups are grouped into "grids" in a similar way. This hierarchy allows adjustment of the kernel invocations to the structure of the data being processed, and also allows scalability of the code. OpenCL defines some barriers in order to synchronize all the work items within a work group: work items belonging to different work groups cannot be synchronized while a kernel is running.

It has been taken into account that each OpenCL device has its own memory hierarchy: a general slow global memory shared across all work groups; fast local memory spaces shared by all the work items within a work group; and there is also private memory which is only visible to each single work item. In order to achieve the best performances, those different memory types must to be used to minimize the read/write access time according to particular OpenCL implementations.

Cross-platform computing with OpenCL has taken on an important role in resolving complex problems in many scientific fields [19–23]. But to our knowledge there are no multi-platform box-counting or other FD related studies that take advantage of this parallel programming model. So in this work we present an efficient OpenCL cross-platform implementation of the box-counting algorithm for calculating the FD.

In the rest of the paper we first describe the box-counting method that we have parallelized. Next, we show how we have implemented both GPU and multi-core CPU target algorithms, detailing how we have optimized each of them. In Sect. 4 we show the timings, speedup and efficiency of our methods. Finally we discuss the results, comparing them with the ones achieved in previous work, and summarize all in the conclusions section.

2 Box-counting algorithm

The box-counting algorithm consists of counting how many square boxes are needed to completely cover a model represented by pixels (2D) or voxels (3D). Each of these boxes is labeled as *black*, *gray* or *white*, depending on if it is completely, partially or not contained by the object. Each square box has an *edge size*, which is equal to the length in pixels/voxels of the side of the box. The process of counting boxes has to be repeated for different edge box sizes. An example of the box-counting process is represented in Fig. 1. In this example, it is shown how the original model (Fig. 1A) is completely covered with boxes with different edge sizes. Thus, for an edge size of 2 pixels (Fig. 1B) the box-counting is 64 *black* boxes, 40 *gray* boxes and 152 *white* boxes; for an edge size of 4 pixels (Fig. 1C) the box-counting is 12 *black* boxes, 28 *gray* boxes and 24 *white* boxes, and for a box edge size of 8 pixels (Fig. 1D) the box-counting is 0 *black* boxes, 12 *gray* boxes and 4 *white* boxes. From these data, the value of the FD for a selected type of voxel (*white*, *gray*, *black*, *black* + *gray*) is calculated through a log–log linear regression, in which the X axis represents the

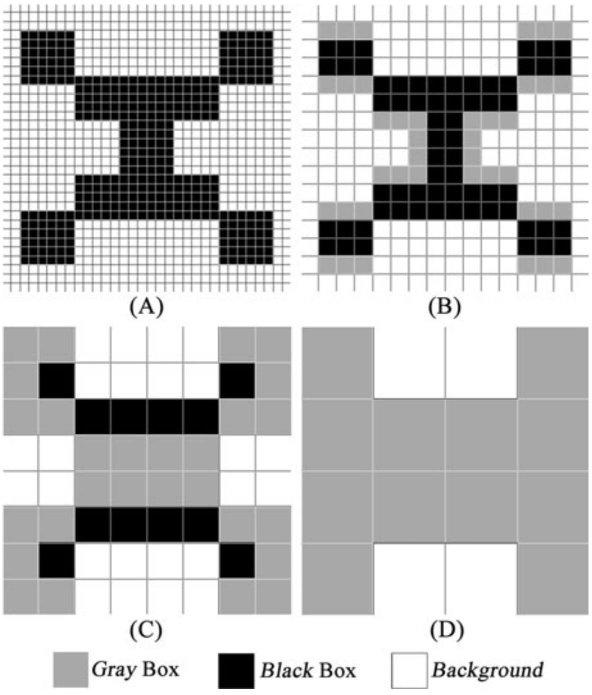


Fig. 1 (**A**) Original model. In (**B**), (**C**) and (**D**) the *black* and *gray boxes* (with different edge sizes) that cover the original model are shown

inverse box size and the Y axis represents the number of boxes for that type of voxel. The FD is estimated as the slope of this linear regression [11].

Hou's box-counting algorithm is an efficient method widely used in the literature. This algorithm has a time complexity of $\mathrm{O}(N \cdot \ln(N))$, and this is the box-counting method that we deal with. In the rest of the paper we assume the 3D case, the 2D case being equivalent but simpler.

Hou's box-counting algorithm directly determines the global position of each voxel of the 3D model through a particular representation of its coordinates. This allows us to easily determine in which box the voxel is contained. The global position of a voxel is uniquely determined through the binary representation of its coordinates with k bits, where 2^k voxels is the edge size of the box that completely covers the model. Grids of boxes are iteratively generated, each one with a different edge size of 2^m voxels, where $0 \le m \le k$. So considering that n is the number of voxels within a box of the grid, this box will be labeled as black if $n = (2^m)^3$, as gray if $0 < n < (2^m)^3$, and finally labeled as white if $n = 0$.

The detailed steps of Hou's box-counting algorithm for the 3D case are:

(A) Initialization: The voxel coordinates x, y and z, with range $0, \dots, 2^k - 1$, must be interpreted as a number in binary representation. Thus each coordinate will be composed of k bits.

(B) Bit Intercalate: The binary coordinates are then intercalated for each voxel, forming a large bit string. The intercalated bit string of a voxel situated in the position $([x_{k-1} \cdots x_2 x_1 x_0]_{\mathrm{bin}}, [y_{k-1} \cdots y_2 y_1 y_0]_{\mathrm{bin}}, [z_{k-1} \cdots z_2 z_1 z_0]_{\mathrm{bin}})$, is $(x_{k-1} y_{k-1} z_{k-1} \cdots x_2 y_2 z_2 x_1 y_1 x_1 x_0 y_0 z_0)_{\mathrm{bin}}$. Thus, the intercalated bit string must have a size equal to $k \cdot 3$ bits. As a result, a list of intercalated bit string is obtained, where each bit string uniquely identifies one voxel.

(C) Sorting: The list of intercalated bit strings is sorted according to the value of each intercalated bit string.

(D) Mask Bit Strings: Finally, $m \cdot 3$ bits on the right of each bit string are then masked to 0. If two or more masked bit strings have the same value, then their corresponding voxels belong to the same box with an edge size of 2^m. As a result of this masking step, the number of distinct values in the list gives us the number of boxes (*black boxes* + *gray boxes*) of edge size 2^m necessary to completely cover the 3D model, thus performing the box-counting for an edge size of 2^m. White boxes are then obtained trivially. We differentiate between *black* and *gray* boxes using the fixed number of voxels required to fill a box. This process is iteratively repeated while $m \leq k$.

After executing all these steps the box-counting has been completed, yielding the number of *black*, *gray* and *white* boxes, for edge size 2^m with $0 \leq m \leq k$, to cover the whole 3D voxelized model.

The previous steps are graphically represented in Fig. 2. In this case, we represent a 3D model formed by 10 voxels (red voxels) inside a 3D grid with a resolution of $k = 2$ and 64 voxels (for the sake of clarity, Fig. 2 shows only 6 empty voxels of the whole grid). In this figure, we only represent the step D for $m = 1$.

There are some implementations of this algorithm on CPU, like the one performed by Kruger in [24], or a recent MATLAB implementation in [5]. But, as mentioned throughout this paper, to our knowledge there are no parallel implementations of Hou's algorithm, except the one developed with NVIDIA CUDA and presented in [18].

The main contribution of this paper is the analysis and development of a cross-platform OpenCL implementation of Hou's box-counting algorithm, for its efficient execution on GPUs and multi-core CPUs. For the GPU case, this requires a hand-on approach for porting the CUDA algorithm to OpenCL, since several problems appear mainly due to the lack in OpenCL of high-optimized auxiliary libraries already available for CUDA. As we have shown, sorting is a key process in the box-counting algorithm, so we need to study, test and select the best available OpenCL sorting algorithms for GPU and for multi-core CPU.

3 Cross-platform box-counting parallelization

As seen before, OpenCL offers the capacity of launching multiple independent work items in a GPU device and/or (simultaneously) in a multi-core CPU, independently of the device vendor. We have developed two optimized implementations: an OpenCL algorithm for execution on a GPU device, and a specific OpenCL approach to exploit the multi-core CPU capabilities.

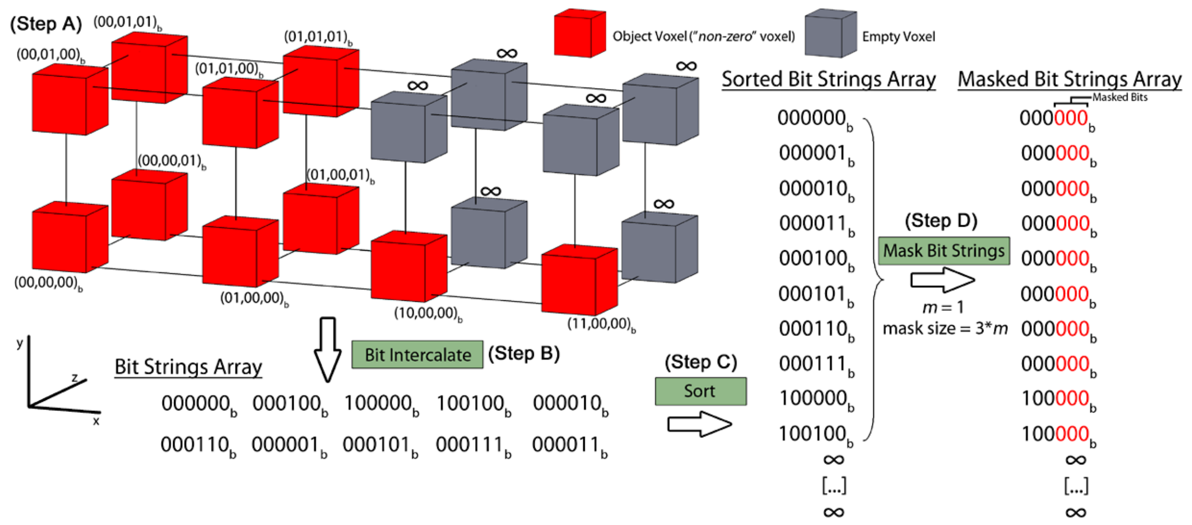


Fig. 2 Hou's box-counting algorithm and its main functions. Step D is represented for $m = 1$ and it is iteratively repeated while $m \leq k$ (Color figure online)

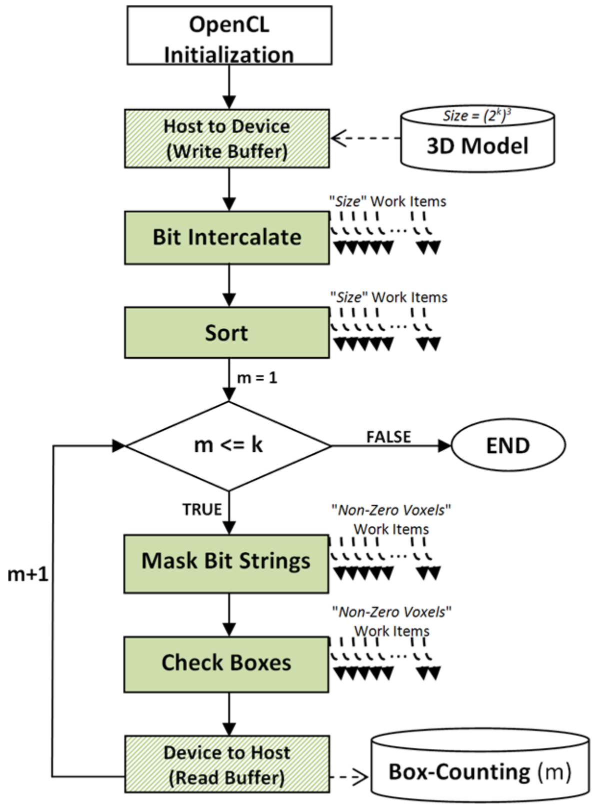


Fig. 3 Data flow chart for the GPU OpenCL box-counting algorithm

3.1 Implementation on GPU

In this section, we describe how Hou's box-counting algorithm has been parallelized in order to improve performance when executing it on GPUs with OpenCL. In our case, OpenCL is accompanied by the C programming language.

First of all, Fig. 3 shows the main modules and data flow of our GPU box-counting algorithm. It is important to note that the shaded functions in Fig. 3 are executed on the GPU side.

As in any OpenCL implementation, the very first step is to initialize the OpenCL context and the target device, or devices, at runtime [17]. At this step, the platform (NVIDIA, AMD or Intel) and the device (GPU or CPU) where the OpenCL algorithm will run on are determined. Due to the cross-platform character of OpenCL, all existing kernel functions must be compiled in execution time, creating a program associated to a concrete device, and then compiling that program for it. Once the program is compiled, individual kernel functions can be extracted and executed in parallel by any C algorithm or function.

Then, once the OpenCL context has been initialized, the dataset (which in our case is a binary array representing the *non-zero* and the *empty* voxels) is transferred from the host side (RAM memory) to the device side (GPU global memory) through the

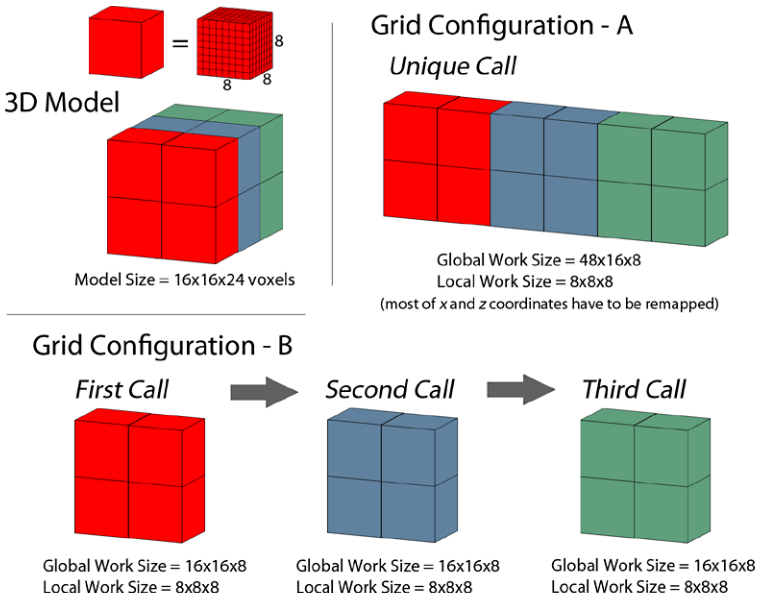


Fig. 4 Grid configuration options for the Bit Intercalate kernel

PCI-express bus, as in any GPGPU application. Once the data is on the GPU, kernels can operate on it.

We created one OpenCL kernel for each main Hou's algorithm step. So we refer to the first kernel as *Bit Intercalate*. This kernel corresponds to the step B introduced in Sect. 2. When programming for the GPU, it is important to test different work group and grid configurations, attempting to minimize the running time of the process. We launch one work item per voxel, so each work item of this kernel accesses to only one position of the 3D model. Work items are grouped into three-dimensional work groups with a size of 8 in each axis, since the original dataset represents a 3D model in our case.

Work groups are contained within bi-dimensional grids, so we can organize the work-groups in two ways (see Fig. 4). The first option (*grid configuration—A*) consists of launching a unique bi-dimensional grid with work groups of work items as necessary to cover the whole 3D model. This configuration implies that the indexes of the work items that do not work over the first 8 slices of the model (since each work group has a size of 8 work items in the Z axis) have to be remapped to fit to the grid of voxels, since there is not a direct correspondence between each work item coordinate and the voxel ones. In the second option (*grid configuration—B*), the *Bit Intercalate* kernel has to be invoked several times to completely operate over the whole 3D model, executing the kernel on a different section of the model each time. As an advantage, no coordinates remapping is necessary in this case, the work item's coordinates directly matches with the voxelized model ones. Therefore, the kernel launched with *grid configuration—A* takes more running time than each kernel of the set launched if option B is selected, since with the latter each work item's coordinate fits perfectly to the 3D model. According to our tests, the set of kernels launched with *grid configuration—B* is around 50 % faster than the one with *grid configuration—A*. However, with *grid configuration—B*, more host time delays are introduced due to the iterative invocation of the kernel. To summarize therefore, the running times of the whole process are good whatever the chosen grid configuration, but we found that by using *grid configuration—A* the final time is better, achieving an improvement of around 3 %.

The *Bit Intercalate* kernel works as follows. If the work item matches with an empty voxel, it stores a predefined and unreachable value in the *bit strings array* (represented as *infinite* in Figures for clarification, $(2^k)^3$ in the code). Otherwise, if the position in the model is not empty (*non-zero* voxel, red cube in Fig. 2), each work item interprets the voxel coordinates (x, y, z) as three binary values: $([x_{k-1} \cdots x_1 x_0]_{\text{bin}}, [y_{k-1} \cdots y_1 y_0]_{\text{bin}}, [z_{k-1} \cdots z_1 z_0]_{\text{bin}})$. Then, those bits are intercalated as shown in Sect. 2, step B. There is also a graphical example of this in Fig. 2 for further understanding. Finally, each work item stores the final intercalated bit string in the *bit string* global memory *array*. So when all the work items end their execution, the array of intercalated coordinates (*bit strings array*) is completely generated.

The next step consists of sorting the *bit strings array* generated by the *Bit Intercalate* kernel. Since sorting data is a fundamental and necessary step in many algorithms, there are some studies that propose sorting methods implemented with OpenCL which take advantage of the GPU characteristics in order to improve performance. Therefore, we have studied these current OpenCL parallel sorting algorithms in order to determine which one is the best, and then included it as a module of our OpenCL-GPU box-counting algorithm. The first OpenCL-based sorting algorithm that we tested is the one included in the NVIDIA OpenCL SDK: the OpenCL Sorting Networks algorithm. This algorithm is based on the Bitonic Sort and is adequate for small datasets. This is not our case as we work with 3D models, so we have a large amount of data. In addition, that algorithm only sorts input arrays with a power of two sizes, which is a limiting factor. Another widely used sorting algorithm based on the Bitonic Sort is the one presented by Bainville [25]. This is a fast and well-optimized sorting algorithm designed for GPUs, but at the moment it is also limited to input datasets of a size power of two. We also test a Radix Sort algorithm presented in [26], which is based on the work presented by Zagha and Blelloch in [27], and improved with some techniques proposed by Satish et al. in [28]. This algorithm is implemented and included on the *clpp* library [29]. This Radix Sort algorithm was presented as the fastest GPU based sorting algorithm but, as we will see later, it has now been outperformed by Bainville's sorting algorithm. However, the Radix Sort algorithm is more general, allowing a dataset to be sorted with an arbitrary number of elements, which is a very important fact. The key is that if in the future a new and better OpenCL-based GPU sorting algorithm is developed, it could be directly introduced into our algorithm, thus improving the general performance of the process.

After this step, the bit strings array is sorted in ascending order, i.e. *Sorted Bit Strings Array* in Fig. 2. In this way, useless bit strings which correspond to empty voxels (represented as infinite in Fig. 2) are then moved to the end of the array.

We then advance to the main loop represented in Fig. 3. As established in Sect. 2, the edge size of the box that completely covers the 3D model is equal to 2^k, for example, if the model has a resolution of 256 voxels, k is equal to 8. So, for $m = 1$ and

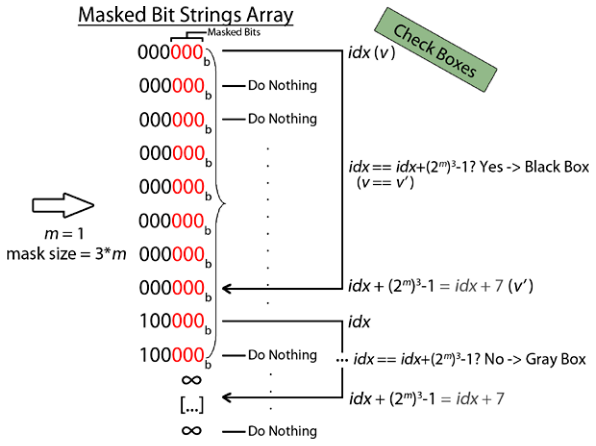

Masked Bit Strings Array
Masked Bits
Check Boxes
000000b
Do Nothing
Do Nothing
m = 1
mask size = 3*m
100000b
∞
[...]
idx (v)
idx == idx+(2^m)^3-1? Yes -> Black Box (v == v′)
idx + (2^m)^3-1 = idx + 7 (v′)
idx
idx == idx+(2^m)^3-1? No -> Gray Box
idx + (2^m)^3-1 = idx + 7

Fig. 5 *Check Boxes* procedure. It starts from an array of masked bit strings, and then it detects which boxes are *black* and which are *gray*. This sample corresponds to Fig. 2

while $m \leq k$ two GPU kernels are iteratively launched: *Mask Bit Strings* and *Check Boxes*. These kernels work on the sorted bit strings array, thus performing the box-counting calculation for incremental box resolutions. Both kernels are launched with the same OpenCL configuration: since both kernels operate on a one-dimensional array, 1D work groups with 512 work items each are launched. These groups are contained in a bi-dimensional grid with as many work groups as necessary. In this case, we launch as many work items as non-zero voxels contained in the 3D model, since it makes no sense that work items perform this processing on the array positions labeled as *infinite*. In this way only real working processes are executed, thus ignoring useless positions of the bit strings array, as previously mentioned. This is also shown in Fig. 2—Step D.

The *Mask Bit Strings* kernel is the simplest one. Each work item is responsible for masking to 0 $m \cdot 3$ bits on the right of a bit string, as explained in the first lines of Sect. 2—Step D. This kernel execution generates the *Masked Bit Strings Array* shown in Fig. 2.

Once the masked coordinates are generated, it must be determined how many groups of equal values are in the array, and also the size of these groups, thus identifying the *black* and *gray* boxes of the box-counting. For this, the kernel called *Check Boxes* is invoked. It determines when a box with an edge size of 2^m is *black* or *gray*. For a better understanding, the *Check Boxes* procedure is graphically detailed in Fig. 5 and the kernel pseudo-code is shown in Listing 1. Figure 5 is the continuation of the process outlined in Fig. 2. Each GPU work item works now as follows: first, it reads a value, v, from the masked bit string array and determines, by reading the preceding position, if v is the first occurrence of a set of repeated values. If not, the work item is not able to count any *black* or *gray* box. These work items are represented as "Do Nothing" in Fig. 5. On the contrary, if the work item has found the first appearance of v, it jumps $(2^m)^3 - 1$ positions in the array (which is the maximum number of voxels within a box, minus 1) and then it reads another value, v'. Due to the properties of the sorted array, we can ensure that if the value of v is equal to the value of v', then a *black* box is detected; if not, one *gray* box is counted.

The last issue, related to the *Check Boxes* kernel, is to determine how to store the number of *black* and *gray* boxes, because race condition hazards could appear since

```
1.  __kernel void CheckBoxes(__global int *intedCoords,int modelsize,
    int size, int boxSize, __global int *d_blacks, __global int
    *d_grays){
2.
3.  __local int blacks[WORKGROUPSIZE]; //WORKGROUPSIZE=512 In our case
4.  __local int grays[WORKGROUPSIZE];
5.
6.  int tid = get_local_id(0);
7.  int idx = get_global_id(0);//Study Voxel ID
8.
9.  if(idx<size){ //Valid array position?
10.       value = intedCoords[idx];
11.
12.       if(idx==0 || value!=intedCoords[idx-1]){//Is the first?
13.             if(idx+(boxSize-1)<modelsize){ //Can jump?
14.                   if(value==intedCoords[idx+(boxSize-1)]){
15.                         blacks[tid]=1; //Store in LocalMem
16.                   }else{
17.                         grays[tid]=1; //Store in LocalMem
18.                   }
19.             }else{
20.                   grays[tid]=1; //Store in LocalMem
21.             }
22.       }
23.  }
24.  barrier(CLK_LOCAL_MEM_FENCE);
25.  if(tid<32){ //Only the first warp.
26.        int totalB=0; int totalG=0;
27.        int offset = (BLOCKSIZE/32);
28.        int gid = get_num_groups(0)*get_group_id(1)+get_group_id(0);
29.        for(int j= offset*tid;j<offset*tid+offset; j++){
30.              //To perform partial count
31.              totalG += grays[j];
32.              totalB += blacks[j];
33.        }
34.        d_blacks[gid*32+tid]=totalB; //Store in GlobalMem
35.        d_grays[gid*32+tid]=totalG; //Store in GlobalMem
36.  }
37. }
```

Listing 1 OpenCL kernel of *Check Boxes* procedure

the work items are executed concurrently. A first naive implementation is to perform atomic additions over a global memory position, but the performance is thus dramatically reduced, since GPU atomic operations require many instruction cycles. On the opposite side, each work item could write one value to global memory, and later the set of written values could be added on to the CPU or a reduction kernel used. This last solution implies many writes to the slow global memory, which is clearly inefficient. We finally implement an intermediate solution, as can be seen in Listing 1. In our *Check Boxes* kernel (Listing 1) we use the fast local memory. Each work item writes in a local memory position if it detects a *black* box, a *gray* box or nothing. When all work items within a work group have finished (line 24 in Listing 1), the first 32 work items (corresponding to the first *warp*) partially add the local memory values stored previously, finally storing the intermediate value in global memory. By applying this strategy a large number global memory writes are avoided. In addition, the global memory write instructions that are finally performed benefit from coalesced memory accesses, since consecutive work items write into consecutive memory positions. Other improvements, like avoiding bank conflicts or minimizing work item divergence, have also been applied in this kernel implementation. When the whole execution of the *Check Boxes* kernel has finished, there are 32 intermediate counters per each work group launched. These counters are finally added and transferred to the host, obtaining the box-counting for boxes with an edge size of 2^m.

We tried to obtain the best performance possible by programming our own GPU kernels and using the tools and libraries that are offered to the OpenCL developers. The structure and procedures of the OpenCL algorithm exposed in this work are very similar to the ones of the CUDA algorithm presented in [18], but there are some differences. First, in the CUDA implementation the array that contains the box-counting results is placed on the GPU side, and is completely transferred to the CPU at the end. Meanwhile, in the OpenCL implementation, the final results array resides on the CPU side. Thus, the data (those 32 intermediate counters previously explained) is transferred step by step at the end of each iteration, and then is summarized and stored on the CPU (Fig. 3). This is because we do not have a very efficient reduction function in OpenCL as we have in CUDA, thanks to the well-optimized Thrust Library [30], so it is preferable to perform that operation on the CPU. The second difference is referred to the launch configuration of the *Bit Intercalate* kernel. As seen at the beginning of this section, we select Fig. 4—*Configuration A* because it offers the better performance; however, in the CUDA case we determine that the best option is Fig. 4—*Configuration B*. Nevertheless, we coded the three kernels in the same way in OpenCL as CUDA, with the necessary syntax changes [31]. Another difference is the OpenCL initialization process outlined at the beginning of Sect. 3.1, which is performed in an implicit way when programming with CUDA. Also, the counting process of non-zero voxels is performed in the CPU in the OpenCL case, while in CUDA it is performed on the GPU through a fine-optimized Thrust's function. In addition, the sorting algorithm used in this implementation is different to the one used with CUDA. This fact will have a great influence on the differences in performance between both algorithms, as will be examined in further detail in the discussion section.

3.2 Implementation on CPU

In this section we describe how Hou's box-counting algorithm has been parallelized in order to achieve optimal performance when executing it on multi-core CPUs.

One of the major theoretical advantages of OpenCL is its platform portability, which allows us to execute the same program on different devices such as multi-core CPUs or GPUs. Therefore, our first multi-threaded implementation consisted of directly launching the OpenCL algorithm described previously, selecting as the processing device a multi-core CPU instead of a GPU. But unsatisfactory results were obtained this way: the box-counting is well calculated, but the running time is dramatically increased. As will be seen in the results section, the OpenCL box-counting algorithm optimized for the GPU is around 30 times slower than the single-threaded implementation when both of them run on the same multi-core CPU. The most fine grain GPU-optimized module is the sorting algorithm, and it is the main cause of that bad result. If a sorting algorithm optimized for the CPU is used, but maintaining the rest of the kernels as equal as they were coded for the GPU, the running time decreases. In this way, this multi-core CPU execution improves the original single-thread implementation, but it achieves poor acceleration rates. All of this is detailed in the results section.

So although OpenCL programs could be directly executed on multiple platforms, if high levels of parallelism are required it is necessary to adapt the code and apply specific-device improvements. It must be considered that GPUs are processors with simple-instruction multiple-data (SIMD) architecture and are ideal for exploiting data parallelism, while multi-core CPUs have multiple-instruction multiple-data (MIMD) architecture and benefit from a task parallelism scheme. Therefore, for the multi-core CPU OpenCL implementation, the code has to be reordered and a different launch configuration set.

We implement a new OpenCL algorithm that uses as much as possible the processing capacity of a multi-core CPU. As seen in the previous section, when ways of achieving optimization are considered for a parallel GPU execution, efforts are centered on minimizing data transfers between devices, reducing as much as possible read/writes from/to global memory, and avoiding diverging execution paths. These optimization points are not fundamental in a multi-core CPU programming model, because (1) the host and the device are the same processor, so there is no need to perform any real transfer between devices, (2) the CPU uses fast RAM memory and has an efficient automatic cache system, and (3) the CPU has a branch predictor, and therefore divergence paths are not such a hindrance.

Figure 6 shows the main modules and data flow of our parallel CPU box-counting algorithm. This flow chart is similar to that presented in Fig. 3, since the same algorithm modules remain. First of all, the OpenCL initialization process works equally to the one described in the OpenCL-GPU algorithm discussion. In this case, the CPU device and a valid OpenCL platform (Intel or AMD) must be selected. From this the OpenCL context and queues are constructed, and then the kernels can be compiled in the same way as when working with a GPU target. The Host-Device copies are represented in Fig. 6, but actually they are more like a type casting (converting a C array into an OpenCL one) than a high-cost copy between different devices, since in this case host and device are the same processor: the CPU.

The main difference resides in the launch configuration of the kernels and also how these kernels are coded. On the one hand, a large number of work items are launched when working in a GPU context, in which each of them work on a minimum set of data. On the other hand, in a multi-core CPU context it is sufficient to launch a few work items that work on several data, thus exploiting the task parallelism character of the hardware. Therefore, in order to launch the algorithm for occupying all CPU cores we split the main procedures in Hou's box-counting algorithm (*Bit Intercalate*, *Mask Bit Strings*, *and Check Boxes*) into *Number-of-Cores* parts. Thus each OpenCL work item will execute each kernel on a fragment of the input dataset instead of over a single data, as occurs in the GPU version.

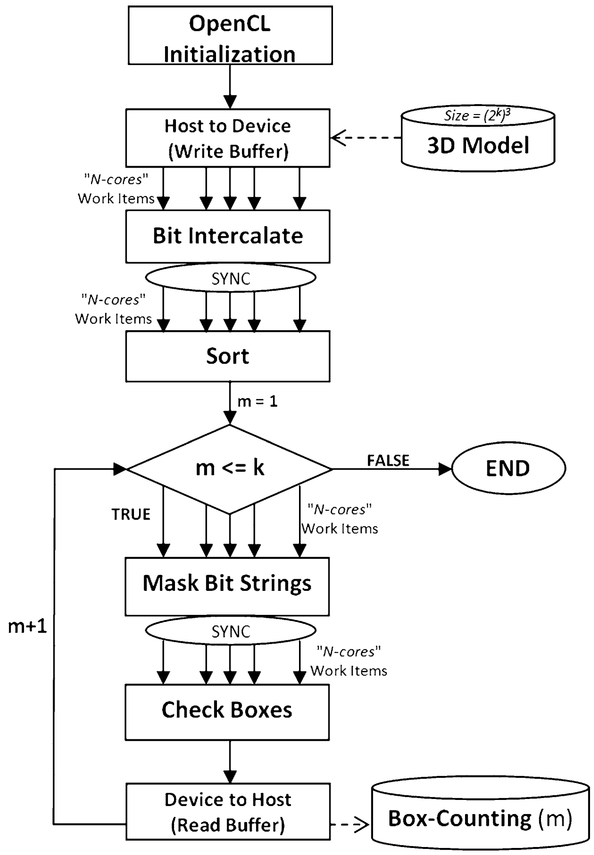


Fig. 6 Data flow chart for the multi-core CPU OpenCL box-counting algorithm

It is clear that when dealing with a multi-core CPU, the amount of physical cores belonging to a specific CPU could provide an estimation of the speedup that will be achieved with the parallelized algorithm. But the partitioning of the input data and also the presence of divergent execution paths could cause an execution balance problem, since it is difficult to ensure that the computational load will be equally distributed along the CPU threads. So if one or more threads have more computational load than the others, the theoretical speedup limit will not be reached.

For example, in the *Bit Intercalate* kernel each work item will face empty voxels and also voxels that actually belong to the 3D model (non-zero voxels). In the first case, the processing is fast and simple since, as seen in previous sections, the work item only has to assign a fixed value in the array. In the second case, the coordinates of the voxel have to be intercalated, a process which has a high computational cost. Unfortunately, it is not possible to ensure that voxels will be equally distributed along the execution threads (unless the model is something like a filled cube), so some threads will work more than others and the theoretical performance peak will not be reached. This will be seen in more detail in the results section.

Something similar happens with the *Check Boxes* kernel. As previously seen, this kernel works over the sorted bit strings array and is launched after the bit strings have

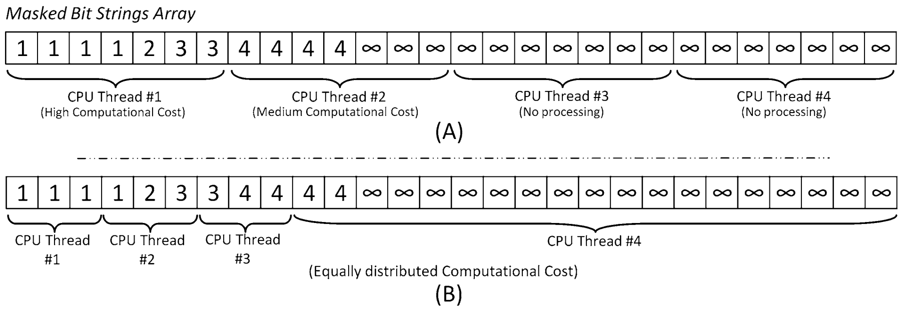


Fig. 7 Data partitioning for the *Check Boxes* kernel on multi-core CPUs. (**A**) With this array partition, the thread #1 has a high computational load, while threads #3 and #4 are idle. (**B**) With this new data distribution, the computational load is almost equally distributed along all the CPU threads, so the whole process is faster than in (**A**)

been masked. In this multi-core CPU implementation, each launched thread has to iterate a part of the masked bit strings array, counting how many equal values exist and thus identifying both *black* and *gray* boxes. If the data of that array is equally divided along the threads most of them will have a low computational load, since they could directly finish and stop their execution when an *infinite* value is found in the array. So in this case the full dataset (the masked bit strings array) does not have to be equally partitioned. It is necessary to apply a more grained division, thus balancing the computational load. This division directly depends on the model, since the amount of data to be distributed along the threads corresponds to the number of non-zero voxels the 3D model has. For example, if the model is a filled cube the whole masked bit strings array has to be equally divided; while in other cases the last positions of the array (positions with a value equal to *infinite*) must be assigned only to the last thread, since this does not imply a remarkable addition to its processing time. An example of this is shown in Fig. 7 for clarification, and a practical experience is outlined in the next section.

When designing this parallel OpenCL implementation we thought of taking advantage of the Intel Streaming SIMD Extensions (SSE) instructions, but finally they were not used for this implementation since they are not interesting in our case. SSE parallel instructions benefit from applying exactly the same arithmetic operations on consecutive data, but our box-counting algorithm has several conditional sentences and not many arithmetic operations. The only function that could fully benefit from using SSE is the one called *Mask Bit Strings*, since in this function the *AND* binary operation is applied to the whole dataset. But if we applied SSE to that function it would imply non-trivial type casting, because the other functions (sorting, bit intercalating and checking *black* and *gray* voxels) need to use standard data types (i.e. *int* instead of *int*4). Anyway, the *Mask Bit Strings* kernel is already very fast without using SSE instructions, so it is not a crucial task. Therefore, we decided not to use the SSE instruction set.

As with the GPU box-counting algorithm, it is necessary to select a sorting algorithm optimized for a multi-threaded CPU execution. We found that there are not many OpenCL sorting algorithms optimized for multi-core CPU. Intel offer in their

OpenCL SDK a parallel Bitonic Sort algorithm [32], but according to our test the performance of this algorithm is not good enough when compared with the Quick Sort algorithm used in our original box-counting implementation. In addition, this algorithm requires an input array of $4 \cdot 2^N$ 32-bit integer items, which is a limitation in the size of the model. We tested a well-known parallel sorting algorithm: the multi-threaded TBB parallel Quick Sort proposed by Intel [33]. This algorithm is very flexible, since it works with random input-data sizes, and provides very good performance, being able to sort around 185 million 32-bit integers per second, according to our tests. So we used this sorting algorithm in our multi-threaded CPU implementation.

4 Implementation results

Once we have shown how the box-counting algorithm has been parallelized on both the GPU and the CPU, a performance analysis is presented in this section, by comparing both parallel implementations with Hou's single-threaded box-counting algorithm.

4.1 Hardware, test models and reference algorithm

In terms of the hardware used to program and measure the performance of the algorithm, our GPU is an NVIDIA GeForce GTX 580 (GTX580), with the NVIDIA driver version 275.33. The GTX580 is based on the NVIDIA Fermi GF100 architecture. GTX580 has 16 multiprocessors, each of them with 32 processors, so the total amount of processors is 512. Each multiprocessor can manage simultaneously a maximum number of independent work items, in this case, 1536. Also, 8 work groups, with up to 1024 work items each, can reside simultaneously in a multiprocessor, but never exceeding the limit of 1536 work items per multiprocessor. Regarding memory, GTX580 has 32K 32-bit registers per multiprocessor (so a theoretical maximum of 21 registers per thread can be used if we can reach full occupancy), an amount up to 48 KB of fast local memory, 16 KB of automatic L1 cache, a total size of 64 KB for constant-memory, and 1536 MB DDR5 of slow global-memory. With respect to the CPU, we used 2 Intel Xeon E5620 @ 2.40 GHz with 4 independent cores each. This was accompanied by 12 GB RAM memory, managed by Windows 7 Professional OS-x64.

Regarding the models used to test the performance of our improved box-counting algorithm, we selected a set of six standard 3D voxelized models with different complexity, features and sizes. This heterogeneous set allowed us to conveniently test the algorithms on different configurations with varying sizes and topologies. These models were: the *Happy Buddha*, *Dragon* and *Bunny* models from the *Stanford 3D Scanning Repository* [34], *Female pelvis* and *Knot rings* obtained in [35], and *Nefertiti* from [36]. Figure 8 shows the voxelization of each model at a resolution of $512 \times 512 \times 512$ voxels.

In order to compare our different tests we saved the final box-counting of each model and the elapsed time of each program. To measure the CPU execution time we

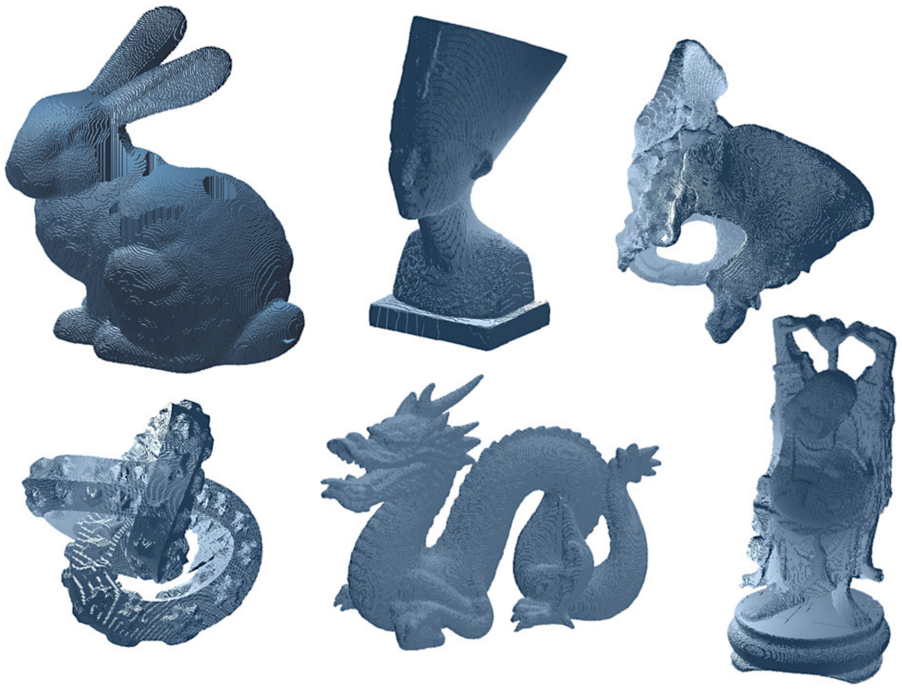


Fig. 8 The six test models at a resolution of 512 × 512 × 512

used the C++ class *PerfTimer* based on *windows.h* library, with microsecond precision.

Regarding the algorithm that we use as reference when analyzing the performance of the parallel algorithms, we coded Hou's single-threaded box-counting algorithm as well as we could, since we could not obtain its original implementation. As the sorting algorithm required by this single-threaded algorithm, we used the fast classic *Quick Sort* C-implementation [37]. In Fig. 9 it can be seen how the processing time is distributed along the whole single-threaded CPU algorithm execution, and how the sort algorithm takes up more than 60 % of the execution, so it is clearly the most time-consuming process of the algorithm, which is a fact that must be taken into account when optimizing and analyzing the performance of the parallel algorithms.

4.2 OpenCL box-counting performance on GPU

First, a general time measure is performed by using the Radix Sort as the sorting algorithm, since it supports an arbitrary size input dataset. Thus, in Fig. 10 the running time of both original single-threaded CPU and OpenCL-GPU algorithms is depicted, together with the speedup ratio achieved.

As previously seen (Fig. 9), the sorting algorithm has a great influence on the final performance of the box-counting algorithm. In this case, the used GPU sorting algorithm (Parallel Radix Sort, as in Fig. 10) takes between 70 % and 80 % of the full session time, a greater percentage than in the single-threaded CPU execution (Fig. 9).

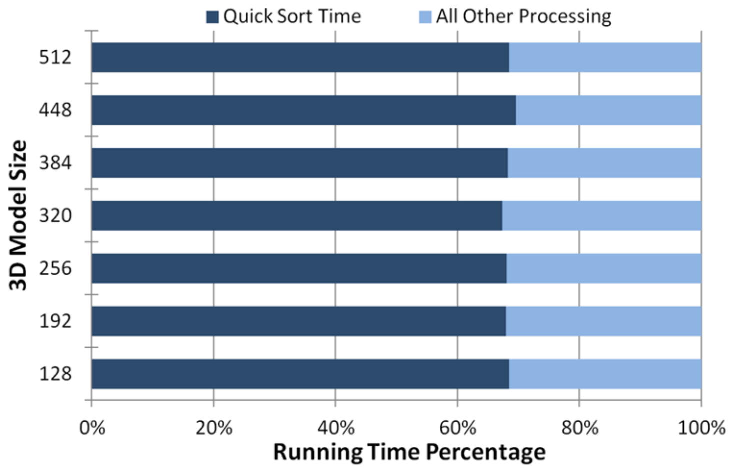


Fig. 9 Running time distribution when executing the original single-threaded algorithm on the CPU for the 3D models at different sizes. Average values for the six test 3D models

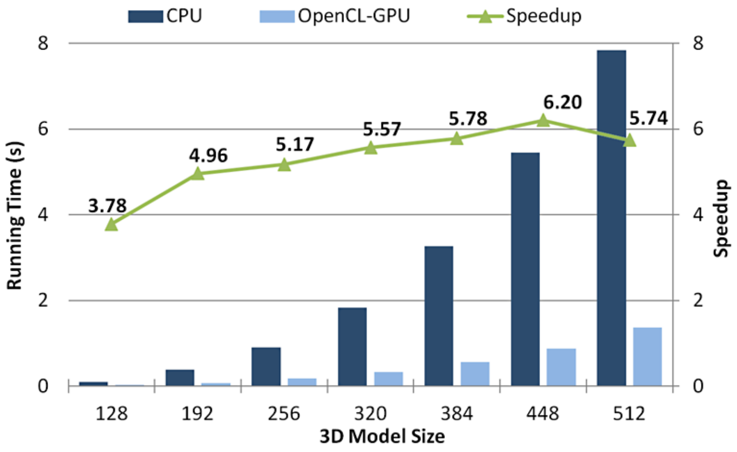


Fig. 10 CPU algorithm (single-threaded) vs. OpenCL-GPU algorithm (multi-threaded). Execution times and speedup (average values for the six test models, with resolutions from 128 × 128 × 128 voxels to 512 × 512× 512 voxels)

This indicates that the remaining functions (GPU kernels in this case) have been further improved than the sorting algorithm, and that the final running time is clearly influenced by the performance of that parallel GPU sorting algorithm. This is evidenced in Fig. 11. The speedup achieved by the Radix Sort GPU sorting algorithm, when comparing it with the original Quick Sort algorithm, is much lower than the speedup achieved by all the other processes. It can be seen how the kernels we implemented improve their performance as the input model size increases, reaching values near to 10× in the best case. This fact demonstrates that our OpenCL kernels (*Bit Intercalate*, *Mask Coordinates*, *Check Boxes*) are well-optimized, but since the sorting algorithm is the most time-consuming process of the algorithm the general speedup

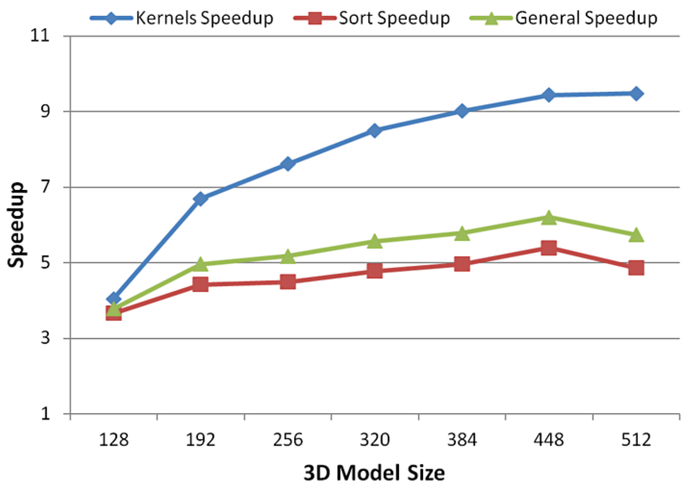


Fig. 11 Radix Sort and kernels isolated speedups. Average values for the six test models

trend is closer to the sorting speedup values. In addition, the sorting algorithm shows a worse performance when the highest model size is selected, around 134 million elements, which is far away to the input sizes tested in [26].

As seen in Sect. 3.1, we highlighted three OpenCL sorting algorithms designed for the GPU: the Radix Sort, Bainville's Bitonic and also NVIDIA's Bitonic Sort algorithm. So we now analyze the isolated performance of these sorting algorithms by measuring their sorting rate. The sorting rate is the number of elements that an algorithm can sort in one second. In our case each element is an integer variable, since each bit string is represented with a 32 bit integer type. The sorting rates achieved are represented in Fig. 12(barchart) in mean terms and for different model resolutions.

In general terms, both Bitonic Sort algorithms show better rates than the Radix Sort when the model size is not too large. In fact, it could be seen how Bainville's Bitonic sort is the one that has the best sorting rate in any case. However, it only works with resolutions that are a power of two, which is an important limit. Meanwhile, the Radix Sort rate remains almost constant whatever the model resolution, while both Bitonic algorithms worsen their performances as the resolution of the model increases. The sorting rate values we achieved are in consonance with others shown in the literature [28, 38].

Figure 12 also summarizes the speedup values achieved by the OpenCL-GPU box-counting algorithm using each of the sorting algorithms. As seen when analyzing the sorting rate, the best algorithm version depends strictly on the resolution of the input 3D model.

The highest acceleration rate achieved was 7.46× with respect to the CPU single-threaded box-counting algorithm, and it was achieved when Bainville's Bitonic Sort algorithm was selected. So our OpenCL parallel box-counting algorithm optimized for the GPU greatly improves the results of the single-threaded original implementation, whatever the resolution of the 3D model.

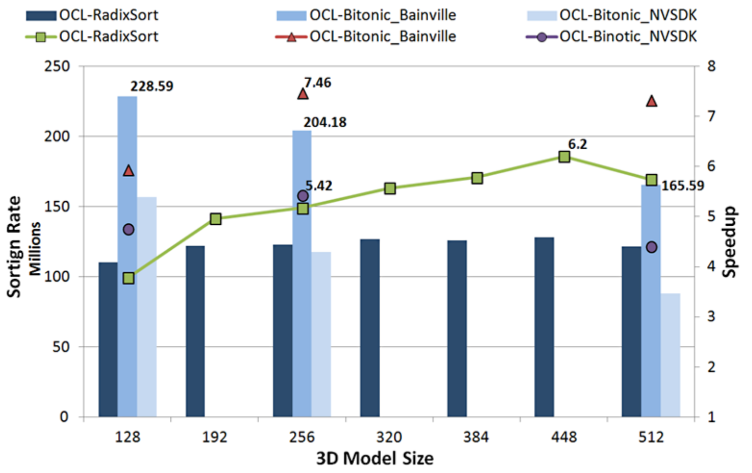


Fig. 12 Sorting rate for each isolated OpenCL sorting algorithm (*barchart*), and box-counting algorithm general speedups depend on the selected sorting algorithm. Average values for all the six test models at different resolutions

4.3 OpenCL box-counting performance on CPU

Our very first test was to launch the OpenCL algorithm designed for GPUs by selecting as the device processor the multi-core CPU. However, the results are very disappointing when compared with the original CPU algorithm. By launching the whole GPU-optimized box-counting algorithm on a multi-core CPU, the running times are, in mean terms, around 30 times worse.

Therefore, the algorithm performance must be improved by applying multi-core CPU oriented specific optimizations, as outlined in Sect. 3.2. First, since the sorting algorithm takes up most of the running time, the GPU-optimized sorting algorithm must be substituted by one designed for execution on multi-core CPUs. We include the fast and efficient parallel Quick Sort algorithm offered by TBB (Threading Building Blocks) [33], the best sorting algorithm we have tested for our specific sorting problem. This TBB sorting algorithm improves the single-threaded Quick Sort algorithm by more than 7× (Fig. 13), which is an expected result since our CPU has 8 cores.

Thus, if the TBB sorting algorithm is integrated together with the rest of the GPU-optimized kernels (*Bit Intercalate*, *Mask Coordinates*, *Check Boxes*) the performance results are more satisfactory in general terms, now improving the single-threaded CPU algorithm running times by 40 % in mean terms, considering the test models and all the resolutions.

But all box-counting GPU-designed kernels could be optimized for a CPU parallel execution, thus improving the general performance of the algorithm. As explained in Sect. 3.2, it is difficult to fully exploit the parallel capabilities of the CPU when parallelizing some functions such as *Bit Intercalate* or *Check Boxes*. Table 1 shows the detailed running times of the *Bit Intercalate* OpenCL kernel when executing it on our CPU (8 cores and 8 threads). We could see how some threads take more time

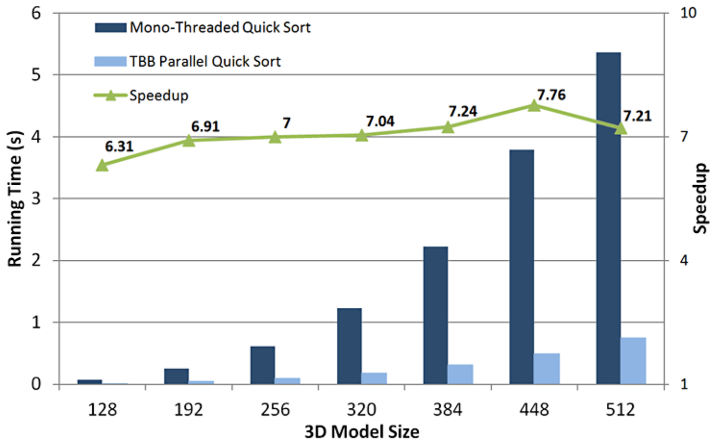


Fig. 13 Single-thread Quick Sort vs. TBB parallel Quick Sort. Average values for the six test models

Table 1 *Bit Intercalate* kernel running times (the higher is italicized). Single-threaded vs. multi-threaded implementation (8 threads). Each model has a resolution of 512 × 512 × 512 voxels. Times shown in seconds

	Bit Intercalate kernel						
	Bunny	Happy Buddha	Nefertiti	Female Pelvis	Knot Rings	Dragon	Fill Cube
#Non-zero voxels	20,669,804	5,269,400	11,205,574	4,942,014	19,494,912	6,456,157	134,217,728
Mono-threaded algorithm	1.654	0.616	1.014	0.595	1.553	0.697	9.139
Thread #1	0.282	*0.149*	0.239	0.076	0.144	0.225	1.510
Thread #2	0.449	0.137	0.195	0.121	0.241	*0.247*	1.525
Thread #3	*0.534*	0.119	0.115	0.155	0.379	0.172	1.529
Thread #4	0.465	0.139	0.134	*0.225*	0.418	0.174	1.562
Thread #5	0.311	0.132	0.186	0.220	0.409	0.113	1.527
Thread #6	0.165	0.124	0.277	0.077	*0.437*	0.062	1.551
Thread #7	0.115	0.102	*0.301*	0.061	0.247	0.066	1.554
Thread #8	0.064	0.072	0.115	0.059	0.115	0.062	*1.575*
Average	0.298	0.122	0.195	0.124	0.298	0.140	1.541
Standard deviation	0.174	0.024	0.072	0.069	0.128	0.075	0.022
Speedup	×3.10	×4.13	×3.37	×2.64	×3.55	×2.82	×5.80

than others, because they have to deal with more non-zero voxels. For example, if we focus on the Bunny model thread number #5 takes eight times more running time than thread number #8. Another example, if we focus on the Rings model we can see how threads number #1, #6, #7, #8 take much less processing time than threads #4 and #5. The problem is that it is not possible to predict in advance where the

Table 2 *Check Boxes* kernel. Running times for the first four iterations. Distribution based on input data vs. distribution based on computational load. The example test model is the *Stanford Bunny* with a resolution of 512 × 512 × 512 voxels

	Check Boxes kernel (*Stanford Bunny* model)							
	Case A: input data equally distributed (i.e. Fig. 8A)				Case B: computational load balance (i.e. Fig. 8B)			
	Iteration #1 (secs)	Iteration #2 (secs)	Iteration #3 (secs)	Iteration #4 (secs)	Iteration #1 (secs)	Iteration #2 (secs)	Iteration #3 (secs)	Iteration #4 (secs)
Thread #1	0.05478	0.02826	0.02333	0.02256	0.00871	0.00444	0.00439	0.00356
Thread #2	0.01285	0.00709	0.00569	0.00547	0.00884	0.00469	0.00367	0.00351
Thread #3	0.00021	0.00005	0.00003	0.00003	0.00892	0.00431	0.00363	0.00350
Thread #4	0.00028	0.00020	0.00003	0.00003	0.00872	0.00440	0.00376	0.00356
Thread #5	0.00024	0.00015	0.00008	0.00003	0.00861	0.00447	0.00374	0.00351
Thread #6	0.00026	0.00012	0.00003	0.00004	0.00887	0.00456	0.00375	0.00356
Thread #7	0.00023	0.00012	0.00003	0.00003	0.00885	0.00458	0.00379	0.00353
Thread #8	0.00029	0.00011	0.00002	0.00003	0.00885	0.00463	0.00367	0.00361
Max – Min	0.05457	0.02821	0.02331	0.02253	0.00031	0.00038	0.00076	0.00011
Standard deviation	0.01916	0.00990	0.00819	0.0079	0.00010	0.00013	0.00024	0.00004

non-zero voxels will be placed, since this depends strictly on the characteristics of the 3D model. Therefore it is not possible to balance the computational load, and the algorithm performance is penalized. However, it can be seen how in the *Fill Cube* example the computational load is equally divided and the speedup is closest to the theoretical maximum. This is because all threads have to work with non-zero voxels, so all threads have to process the same instructions.

Something similar happens with the Check Boxes procedure, but in this case we can avoid the problem by modifying the data partitioning, as previously outlined in Sect. 3.2—Fig. 7. Table 2 shows the detailed running times of each thread with and without that input data redistribution. We can see how in case A (without load redistribution) threads number #1 and #2 always have the greater running times while the other threads are almost idle. Meanwhile, when the computational load is equally distributed (case B), the entire threads take the same running time, thus improving the algorithm performance. This distribution depends strictly on the number of non-zero voxels that the 3D model has.

With these improvements we measure the whole running time of the OpenCL-CPU box-counting algorithm using the TBB multi-threaded Quick Sort. The general performance results are shown in Fig. 14. The general speedup is close to 4×, which is a good rate but not close to the theoretical maximum achievable by our CPU. The processing time distribution for the multi-threaded CPU algorithm is different when compared with the GPU implementation one. In this case, the OpenCL kernels, together with the rest of the algorithm, take more running time than the sorting algorithm.

We coded a new multi-threaded box-counting algorithm designed for multi-core CPUs, maintaining the TBB multi-threaded quick sort algorithm but using the C *pro-*

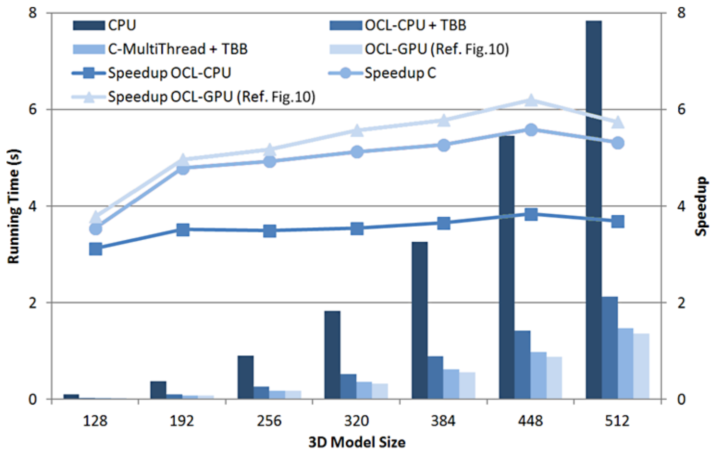


Fig. 14 CPU algorithm (single-threaded) vs. OpenCL-CPU algorithm (multi-threaded), C multi-threaded CPU algorithm (both including the TBB parallel sorting algorithm) and OpenCL-GPU (as shown in Fig. 10). Execution times and speedup (average values for the six test models, with resolutions from 128 × 128 × 128 voxels to 512 × 512 × 512 voxels)

cess.h library [39] (this library is included in Microsoft Visual Studio), following the same structure and philosophy as the OpenCL-CPU algorithm. The results obtained are also shown in Fig. 14. By using this C library better speedups are achieved, although the maximum theoretical rates are not reached. This is mainly due to the single-threaded parts of the algorithm and also to the previously discussed issues related to the unbalanced computational load in some procedures.

5 Discussion and conclusions

We have presented a study about developing a parallel cross-platform implementation of Hou's Box-Counting algorithm using OpenCL. We have detailed how each step of the original algorithm has been implemented, parallelized and optimized for its execution on both the GPU and the multi-core CPU. Related to this, we have outlined the main current OpenCL sorting algorithms. We have focused on the 3D case, because computing the box-counting of large amounts of 3D data is a highly time-consuming process. Nevertheless, our algorithm could easily be applied to the 2D case. Currently, to our knowledge there is no OpenCL implementation of the box-counting algorithm, either for GPUs or multi-core CPUs.

It is important to note that, thanks to the properties of the initialization and execution process of any OpenCL application, both GPU and multi-core CPU implementations could easily be packed in the same redistributable. In this way, the kernels could be dynamically selected according to the selected target device. Thus, we are able to achieve the optimal performance in any case.

With our OpenCL GPU implementation, a final average speedup of more than 7× is achieved when compared with the single-threaded CPU implementation. If the results presented in this paper are compared with the ones that figure in [18], our previous work, it can be seen how the CUDA implementation is approximately 4 times faster in the worst case. This is mainly due to two facts. First, it must be taken into account that nowadays CUDA is a more established technology than OpenCL; therefore, there are a large number of well-optimized libraries that permit us to easily launch classical functions, such as reductions or summations, which fit perfectly on the GPU, thus improving the performance and simplifying the algorithm flow chart.

Regarding the second fact, as seen throughout this paper, Hou's box-counting algorithm is strongly influenced by its sort procedure, so the selected sort algorithm will establish the top speedup that could be reached by the general process. When programming with CUDA, we found a very fast sorting algorithm designed by Merrill et al. [40] and included in the Thrust library [30]. With this algorithm, an impressive performance is reached: a mean sorting rate of 810 million items per second which increase up to 1,000 million in the best case, according to our tests. If we compare these rates with the ones shown in Fig. 12, we can see how that CUDA sort algorithm is between 3 and 5 times faster than the OpenCL ones, depending on the model. This clearly influences the performance differences between both GPU implementations. In fact, if we compare the isolated running times and speedup reached by the GPU kernels that we have implemented we realize that both CUDA and OpenCL kernel performances are good and almost equal between them.

Interesting work to be performed in the future is to implement Merrill's Radix Sort algorithm, included in the Thrust library, with OpenCL. Thus, the performance of several OpenCL applications, including our box-counting implementation, could be improved.

In spite of the apparently low performance when comparing the OpenCL GPU algorithm with our previous CUDA implementation, the one presented in this paper is still interesting and useful. As an example of this, we compare our results with the ones presented in a recent work by de Souza et al. in which Hou's box-counting algorithm is implemented in Matlab in a fast way [5]. To measure the runtime they use the theoretical 3D fractal "Menger sponge" with a resolution of 243 voxels in each axis. We execute our OpenCL GPU implementation using that "Menger sponge" fractal as the input data. While their Matlab program takes 16.49 seconds on a CPU very similar to ours, our parallel OpenCL GPU implementation performed the box-counting calculation in 0.153 seconds, which supposes a performance improvement of 107.77×. If we compare this "Menger sponge" time obtained by the OpenCL algorithm with the one obtained with CUDA, the improvement of the last algorithm is only around 1.78×.

Regarding the multi-core CPU running times, an average speedup of around 4× is achieved by executing the OpenCL box-counting algorithm designed for multi-core CPUs. These OpenCL performance results are below the ones obtained with a classical multi-threading C library, although close to them. OpenCL is a more flexible programming platform, allowing us to set and modify the threads launch configuration in an easier way, but by using a low-level library for multi-threading better speedup rates could be achieved in this case. Although one of the theoretical advantages of OpenCL is that the same code could be executed on different devices (i.e. CPUs and GPUs), we have shown how if we want to reach maximum speedups, the algorithm must be designed taking into consideration the target device.

Acknowledgements This work has been partially supported by the University of Jaén, the Caja Rural de Jaén, the Andalusian Government and the European Union (via ERDF funds) through the research projects UJA2009/13/04 and PI10-TIC-5807.

References

1. Esteban FJ, Sepulcre J, Ruiz de Miras J, Navas J, de Mendizábal NV, Goñi J, Quesada JM, Bejarano B, Villoslada P (2009) Fractal dimension analysis of grey matter in multiple sclerosis. J Neurol Sci 282:67–71
2. Wu YT, Shyu KK, Jao CW, Wang ZY, Soong BW, Wu HM, Wang PS (2010) Fractal dimension analysis for quantifying cerebellar morphological change of multiple system atrophy of the cerebellar type (MSA-C). NeuroImage 49:39–551
3. Shyu KK, Wu YT, Chen TR, Chen HY, Hu HH, Guo WY (2011) Measuring complexity of fetal cortical surface from MR images using 3-D modified box-counting method. IEEE Trans Instrum Meas 60:522–531
4. Kotowski P (2006) Fractal dimension of metallic fracture surface. Int J Fract 141(1–2):269–286
5. de Souza J, Rostirolla SP (2011) A fast MATLAB program to estimate the multifractal spectrum of multidimensional data: application to fractures. Comput Geosci 37(2):241–249
6. Khanbareh H, Wu X, Van der Zwaag S (2012) Analysis of the fractal dimension of grain boundaries of AA7050 aluminum alloys and its relationship to fracture toughness. J Mater Sci 47(17):6246–6253
7. Vahedi A, Gorczyca B (2011) Application of fractal dimensions to study the structure of flocs formed in lime softening process. Water Res 45(2):545–556
8. Khoury M, Wenger R (2010) On the fractal dimension of isosurfaces. IEEE Trans Vis Comput Graph 16:1198–1205
9. Russel D, Hanson J, Ott E (1980) Dimension of strange attractors. Phys Rev Lett 45:1175–1178
10. Ruiz de Miras J, Villoslada P, Navas J, Esteban FJ (2011) UJA-3DFD: a program to compute the 3D fractal dimension from MRI data. Comput Methods Programs Biomed 104:452–460
11. Hou X, Gilmore R, Mindlin GB, Solari HG (1990) An efficient algorithm for fast O($N \cdot \ln(N)$) box counting. Phys Lett A 151:43
12. Liebotich LS, Toth T (1989) A fast algorithm to determine fractal dimension by box counting. Phys Lett A 141:386
13. Bauer W, Mackenzie CD (2001) Cancer detection on a cell-by-cell basis using a fractal dimension analysis. Acta Phys Hung, Heavy Ion Phys 14(1–4):43–50
14. Koster M, Hannawald J, Brameshube W (2006) Simulation of water permeability and water vapor diffusion through hardened cement paste. Comput Mech 37(2):163–172
15. Diaz J, Munoz-Caro C, Nino A (2012) A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans Parallel Distrib Syst 23(8):1369–1386
16. NVIDIA GPU computing documentation (2011). http://developer.nvidia.com/nvidia-gpu-computing-documentation
17. Khronos OpenCl Working Group (2010) The OpenCL specification. Version 1.1. http://www.khronos.org/opencl/
18. Jiménez J, Ruiz de Miras J (2012) Fast box-counting algorithm on GPU. Comput Methods Programs Biomed 108(3):1229–1242
19. Escalera S, Puig A, Amoros O, Salamó M (2011) Intelligent GPGPU classification in volume visualization: a framework based on error-correcting output codes. Comput Graph Forum 30(7):2107–2115
20. Weber R, Gothandaraman A, Hinde RJ, Peterson GD (2011) Comparing hardware accelerators in scientific applications: a case study. IEEE Trans Parallel Distrib Syst 22:58–68
21. Choudhary NK, Navada S, Ginjupalli R, Khanna G (2011) An exploration of OpenCL on multiple hardware platforms for a numerical relativity application. In: Proceedings of the international conference on parallel and distributed computing and systems, pp 87–92
22. Yuan Z, Si W, Liao X, Duan Z, Ding Y, Zhao J (2012) Parallel computing of 3D smoking simulation based on OpenCL heterogeneous platform. J Supercomput 61:84–102
23. Zavala-Romero O, Meyer-Baese A, Meyer-Baese U (2012) Multiplatform GPGPU implementation of the active contours without edges algorithm. In: Proceedings of SPIE, vol 8399
24. Kruger A (1996) Implementation of a fast box-counting algorithm. Comput Phys Commun 98:224–234

25. Bainville E (2011) OpenCL sorting. http://www.bealto.com/gpu-sorting_intro.html
26. Ha L, Krüger J, Silva CT (2009) Fast four-way parallel radix sorting on GPUs. Comput Graph Forum 28(8):2368–2378
27. Zagha M, Blelloch GE (1991) Radix sort for vector multiprocessors. In: Supercomputing'91: proceedings of the 1991 ACM/IEEE conference on supercomputing, New York, NY, USA, 1991, pp 712–721. ISBN: 0818621583
28. Satish N, Harris M, Garland M (2009) Designing efficient sorting algorithms for manycore GPUs. In: IPDPS 2009—proceedings of the 2009 IEEE international parallel and distributed processing symposium
29. clpp—OpenCL Data Parallel Primitives Library (2011). http://code.google.com/p/clpp/
30. Hoberock J, Bell N (2012) Thrust: a parallel Template Library. v1.6.0. http://thrust.github.com/
31. Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J (2012) From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. Parallel Comput 38(8):391–407
32. Intel OpenCL Bitonic Sort algorithm (2011). http://software.intel.com/en-us/articles/vcsource-samples-bitonic-sorting/
33. Intel Threading Building Blocks (TBB) (2008). http://threadingbuildingblocks.org/
34. Stanford university (2011) The Stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep
35. Aim@shape repository (2011). http://shapes.aimatshape.net
36. 3DVIA repository (2011). http://www.3dvia.com
37. QuickSort. http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm
38. Khan FG, Khan OU, Montrucchio B, Giaconne P (2011) Analysis of fast parallel sorting algorithms for GPU architectures. In: Proceedings—2011 9th international conference on frontiers of information technology, FIT 2011, pp 173–178
39. Process.h C library specification. http://www.digitalmars.com/rtl/process.html
40. Merrill D, Grimshaw A (2011) High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. Parallel Process Lett 21:245–272