Parallel Implementation of the Box Counting Algorithm in OpenCL

Ramakrishnan Mukundan

Department of Computer Science and Software Engineering

University of Canterbury

Christchurch, New Zealand.

mukundan@canterbury.ac.nz

ABSTRACT

The Box Counting algorithm is a well-known method for the computation of the fractal dimension of an image. It is often implemented using a recursive subdivision of the image into a set of regular tiles or boxes. Parallel implementations often try to map the boxes to different compute units, and combine the results to get the total number of boxes intersecting a shape. This paper presents a novel and highly efficient method using OpenCL kernels to perform the computation on a per-pixel basis. The mapping and reduction stages are performed in a single pass, and therefore require the enqueuing of only a single kernel. Each instance of the kernel updates the information pertaining to all the boxes containing the pixel, and simultaneously increments the box counters at multiple levels, thereby eliminating the need for another pass to perform the summation. The complete implementation and coding details of the proposed method are outlined. The performance of the method on different processors are analysed with respect to varying image sizes.

Keywords: Fractal dimension; OpenCL kernels; Box counting algorithm; Multifractal analysis; Parallel implementations.

1. Introduction

Fractal analysis has recently found several applications in the field of medical image processing [1]-[3]. The highly complex, but statistically self-similar texture features found in medical images motivated the development of fractal based algorithms for the extraction of diagnostically significant information. Fractal characteristics of tissue images have also been used as texture features for automatic classification [4], [5]. Similarity measures formed using these features were found useful in the classification of pathological cases and the identification of regions of interest [6], [7]. A common algorithmic step in most of the above applications is the computation of the fractal dimension. The fractal dimension is often approximated as the box-counting dimension [8], [9]. Even though the box counting algorithm is a very popular method for estimating the fractal dimension, it is computationally slow as it requires a recursive subdivision of the image into regular grids with varying sizes.

The computation of the fractal dimension also forms the core of many multifractal analysis techniques used in biomedical imaging [10], [11]. Tissue images can be thought of as a superposition of several disjoint textures (alfa-slices), each with its own fractal characteristics. An alfa-slice represents the collection of pixels in a tissue image that satisfy similar scaling laws for some intensity based measure defined over the image [12]. Such a decomposition of an image is found to be useful in both segmentation and classification. The determination of the multifractal spectrum of an image requires the computation of the fractal dimension of all alfa-slices, which often consists of a large number of images. Applications involving the processing of very large images (eg., very high magnification digital microscopy images of biopsy samples) of Gigabyte sizes are also becoming common in the field of medical image processing. For an excellent review of applications of fractal and multifractal methods in the field of medical image analysis, please refer to [13].

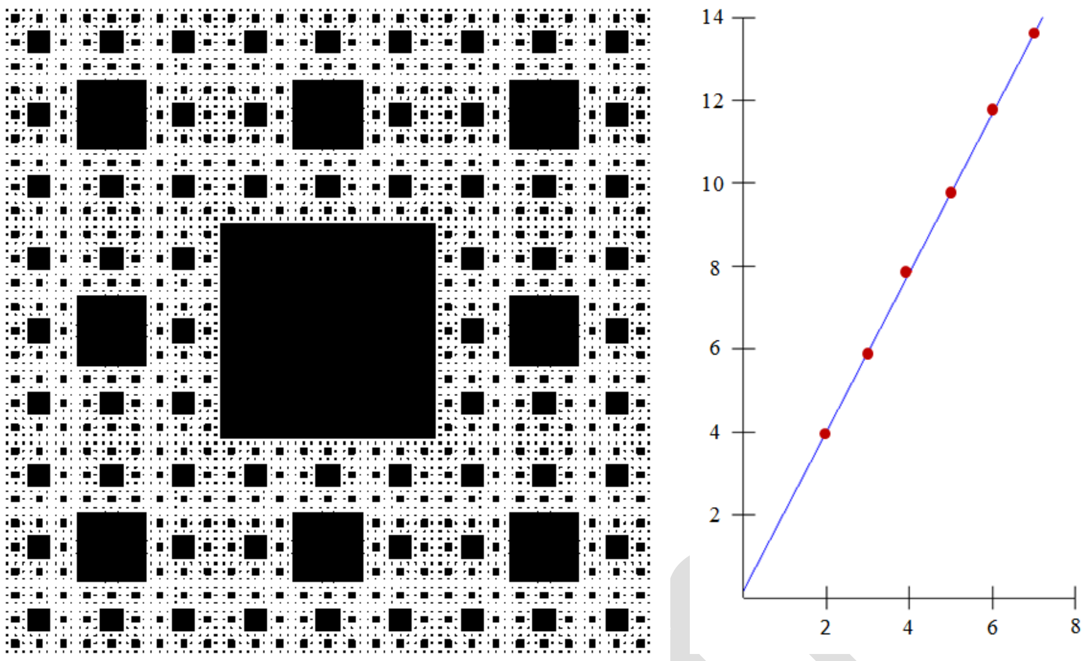
In order to meet the increasing demand for fast fractal computation of either large size or large number of images, recent research has primarily focussed on methods to reduce the complexity at both algorithm level and/or implementation level. Algorithms such as the Higuchi dimension [14] and correlation fractal dimension [15] have been found to be faster than the box counting method. However when the input data set becomes voluminous in size, new computational paradigms involving parallel or concurrent processing are often utilized. Only very recently, some significant efforts have been made in this direction where GPU implementations have been used for computing the fractal dimension [16], [17].

The Open Computing Language (OpenCL) [18] provides an excellent framework for implementing tasks that can be individually computed on rectangular image tiles and then merged to form the final solution. It is therefore ideally suited for computing the fractal dimension of images using the box counting method, but to the author's knowledge, such implementations have not been discussed much in literature. The nVidia developer zone [19] also does not provide any examples of kernels for computing fractal dimension.

This paper provides a detailed description of two implementations of the box counting method using OpenCL kernels. The first approach uses only the global memory to store the box status values at each level, while the second uses the local work-group memory for small box sizes and a reduced amount of global memory for large box sizes. In both cases, the box counts are obtained in the same pass without requiring an additional kernel. The performance improvements and memory requirements are compared and analysed with respect to image sizes varying from 256×256 to 8192×8192 pixels. The paper is organised as follows. The next section gives a set of results that are used as bench marks, obtained using the conventional box counting method. Section 3 introduces the first kernel and discusses its implementation aspects. A second OpenCL kernel is presented in Section 4 along with a comparative analysis of the performance. Conclusions and future work are outlined in Section 5.

2. The Box Counting Algorithm

The well-known box counting algorithm is commonly used as a close approximation of the Hausdorff dimension [8] which is defined based on a covering of a fractal using open discs. In box counting, a regular grid is superimposed on the fractal and the number of cells that intersects the shape is counted. The fractal dimension represents the power-law variation of the total number of cells containing the fractal with respect to the cell size. The method is often implemented using a recursive structure or a nested loop where the grid size is varied in powers of 2 (typically 4, 8, 16, etc., up to some maximum, say 128). A log-log plot of the number of cells intersecting the fractal versus box size ratio gives the dimension as the slope of the best-fitting line. As an example, the fractal known as the Sierpinski Carpet is shown in Fig. 1(a), and its linear regression line in Fig. 1(b). The *x*-axis of the regression plot represents $\log_2$(BoxSize/ImageSize), and the *y*-axis $\log_2$(BoxCount). In this example, the image size is chosen as 512. The first data point corresponds to a box size of 128 (with a *x*-value 2) where the number of boxes intersecting the fractal is 16 (with a *y*-value 4). The computation of the slope of the linear regression line uses only a small amount of data points, and does not contribute to the complexity of the algorithm. Only the core box counting method is therefore discussed in the following sections. The presented methods do not depend on the actual fractal or shape characteristics of the image, and hence it suffices to use only one fractal shape with varying image sizes for the validation and analysis of the methods.



(a) (b)

Figure 1. The Sierpinski Carpet fractal and its linear regression graph.

Fig. 2. gives the variation of average elapsed time for the box counting algorithm as the fractal size is varied from 256×256 to 8192×8192, each time scaling up the size by a factor of 2. For convenience, the *x*-axis is plotted on a logarithmic scale. This figure compares the time taken on two independent processors. The first processor used was an Intel i7-3770 CPU at 3.5 GHz with 8 GB RAM, and its graph is shown as a solid line. The second processor was a laptop with Intel i5-3337U CPU at 1.8 GHz with 8 GB RAM, and its graph is shown as a dotted line. As can be seen from the graphs, large image inputs can take several seconds for the execution of the box counting method, even on fast processing devices.

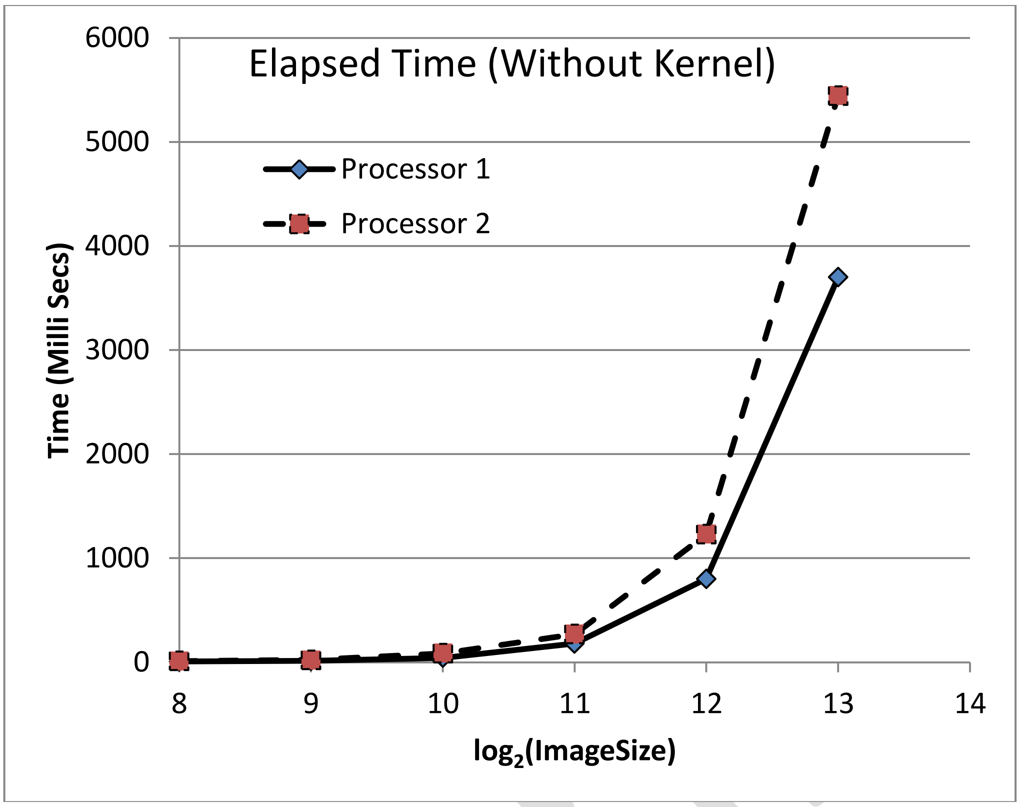


Figure 2. Variation of elapsed time with image size for the box counting algorithm.

3. OpenCL Kernel - 1

The first attempt at parallelizing the box counting algorithm aims at using a single kernel that executes a work item for each pixel of the image. The position of a pixel is used to update the status of all enclosing boxes at different levels, and also to simultaneously update the box counts at those levels. This processing method eliminates the need for another pass (typically using another kernel) for counting the number of boxes. The main data structure required here is an array of bit values representing the status of boxes (showing whether they are intersected by the fractal or not). This array is referred to as "`boxStatus`". The update algorithm is depicted in Fig. 3., using one level of 16 boxes. The coordinates of the pixel and the box size are used to compute the box index in the array. If the value at that location is 0, it is updated to 1. The box counter also is simultaneously incremented. If the value at the location was previously updated to 1 by another pixel within the box, then no action is taken.

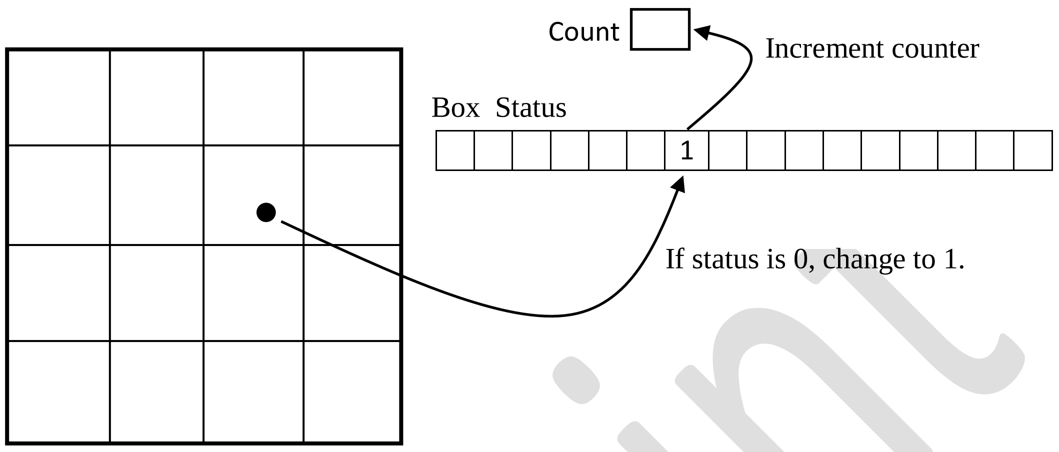


Figure 3. A per-pixel algorithm for updating the box status and box count.

The `boxStatus` array must be accessible to all instances of the kernel that execute on a per-pixel basis, and is therefore created in the global memory. However when multiple work items try to read and update a global memory location, it is essential to have a memory lock mechanism by which an update operation cannot be interrupted. This mechanism is provided by atomic operations in OpenCL [20]. The `atomic_cmpxchg` function is ideal for the operation of comparison and setting of a location in the `boxStatus` array. Even though just a bit array is sufficient for this purpose, the function `atomic_cmpxchg` supports only integer operations. Therefore, `boxStatus` is defined as an integer array. Similarly, incrementing the box count also is done using the atomic function `atomic_inc`.

This `boxStatus` array can be extended to include all boxes at different levels. The box count field also gets extended to an array whose size is the number of levels used in the linear regression model. Consider 6 levels of box sizes 4, 8, 16, 32, 64, and 128 as previously shown in Fig. 1(b). If the image is of size $N \times N$ pixels, then the total size of the `boxStatus` array is given by

$$B = \left(\frac{N}{4}\right)^2 + \left(\frac{N}{8}\right)^2 + \left(\frac{N}{16}\right)^2 + \left(\frac{N}{32}\right)^2 + \left(\frac{N}{64}\right)^2 + \left(\frac{N}{128}\right)^2 = \frac{1365\,N^2}{2^{14}} \qquad (1)$$

Thus, for a 512×512 image, the array has 21840 elements. The global memory requirement increases exponentially with image size, and this is the main drawback of this method. In the next section, an improved solution using local workgroup memory is presented. The variation of the size of the `boxStatus` array with respect to the image size is shown as the solid line plot in Fig. 4.

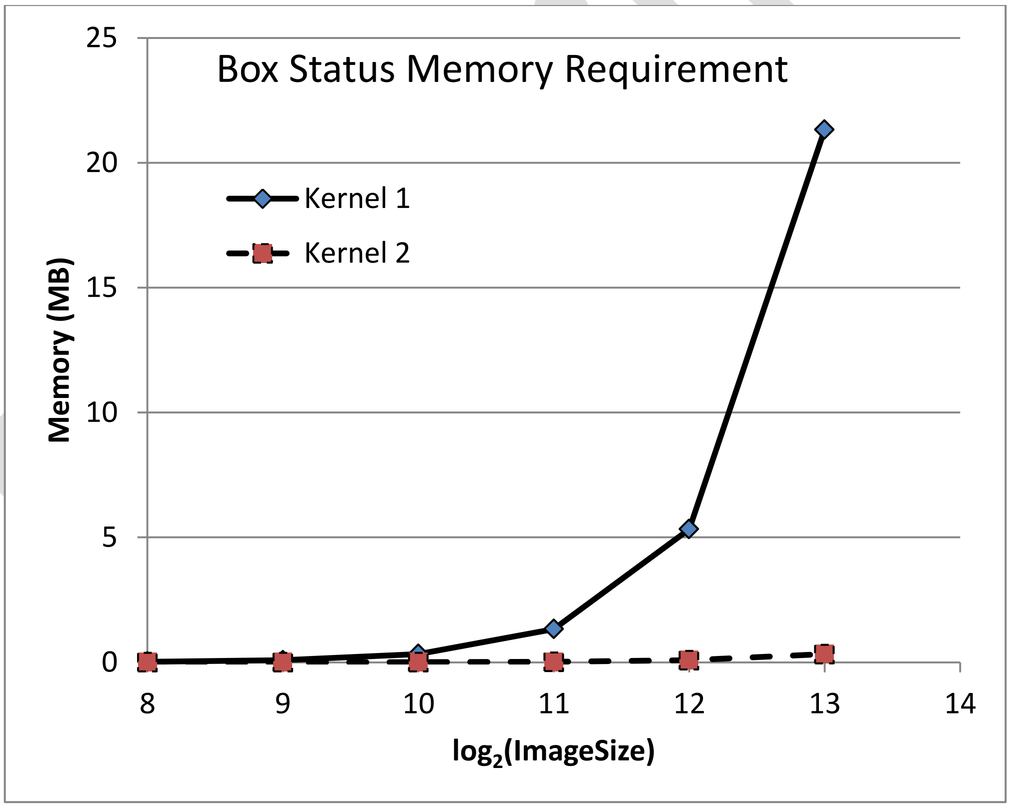


Figure 4. A plot showing the variation of the size of `boxStatus` array with image size.

The kernel code given in Fig. 5 shows the main implementation aspects of the method. Each kernel instance processes a single pixel, and if the pixel value is less than a threshold, the work item immediately exits. Otherwise, the box IDs are obtained from the pixel coordinates and the box size, and the status and count arrays are updated if required, for all six levels.

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                               CLK_ADDRESS_CLAMP |
                               CLK_FILTER_NEAREST;

__kernel void fractal_dimn
                (read_only image2d_t img,
                 __global int* boxStatus,
                  __global int* boxCount )
{
   int ix = get_global_id(0);
   int iy = get_global_id(1);
   int wid = get_image_width(img);

   int iu = 0, iv = 0, offset = 0;
   int index = 0, nboxes = 0, bcindx = 0;
   float4 pixel = read_imagef(img, sampler, (int2)(ix, iy));
   if(pixel.x < 0.5) return;

   for(int boxsize = 4; boxsize <= 128; boxsize <<= 1)
   {
       nboxes = wid / boxsize;
        iu = (ix / boxsize);
        iv = (iy / boxsize);
       index = iv*nboxes + iu + offset;
        if(atomic_cmpxchg(&boxStatus[index], 0, 1) == 0)
                 atomic_inc(&boxCount[bcindx]);
       offset += (nboxes * nboxes);
       bcindx++;
    }
}
```

Figure 5. Kernel code for the box counting algorithm (Implementation 1).

The OpenCL device type was selected as GPU, and the time measurements taken using the high-resolution performance counter given by the `QueryPerformanceCounter` function. The performance improvement provided by the above implementation can be clearly seen by comparing the graphs in Figs. 2 an 6. The two graphs in Fig. 6 correspond to the two processors described in Sec. 2. The desktop machine (Processor 1) had a nVidia

GeForce GTX 650 graphics card (384 CUDA Cores) with 1100 MHz base clock. The laptop (Processor 2) had a nVidia GeForce 710M graphics card (96 CUDA Cores).

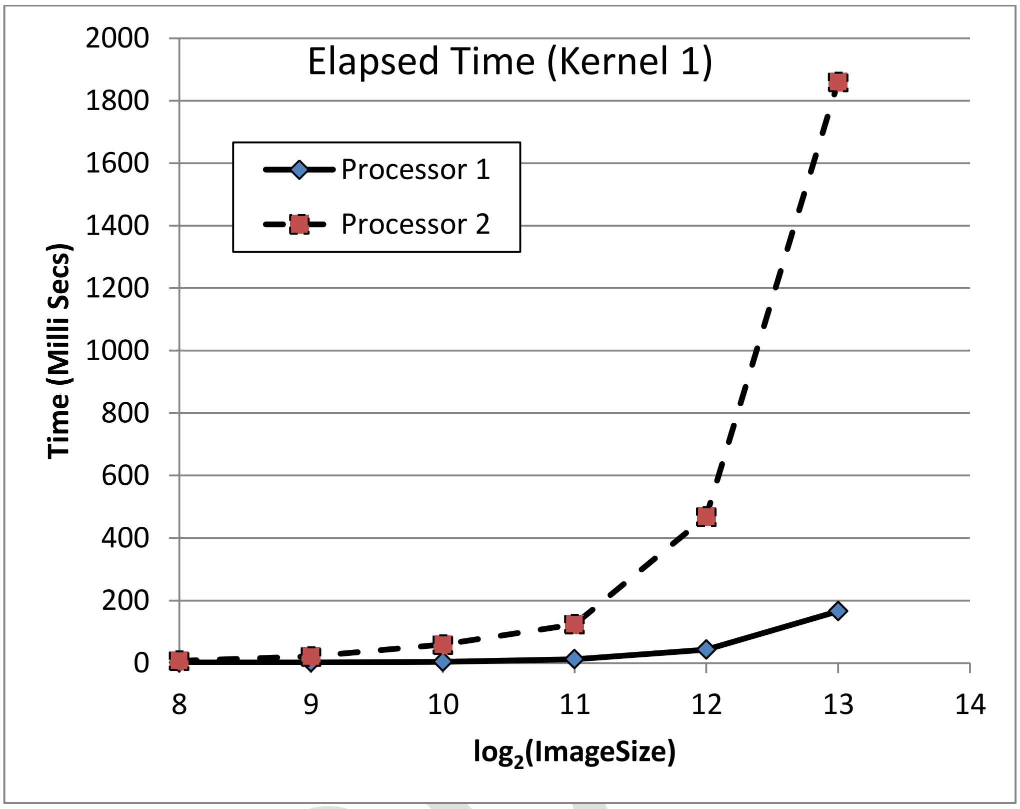


Figure 6. Elapsed time for the execution of the box counting algorithm (Implementation 1) on the GPU of two processors.

4. OpenCL Kernel - 2

The primary limitation of the kernel in the previous section is the requirement for a large global memory. The power of OpenCL work groups can be leveraged in reducing the use of global memory, at the same time improving performance. The main idea here is that the status of all boxes with size smaller than the work group size can be stored in the local memory of the work group. The global memory will then be required only for box sizes that are larger than the work group size. This scheme partitions the `boxStatus` array into a collection of small local memory arrays (`boxStatus_L`), and one global memory array (boxStatus_G) with a significantly reduced size compared to the previous method. OpenCL's execution and memory models for processing two-dimensional data are ideally suited for this type of partitioning of the boxStatus array as shown in Fig. 7.

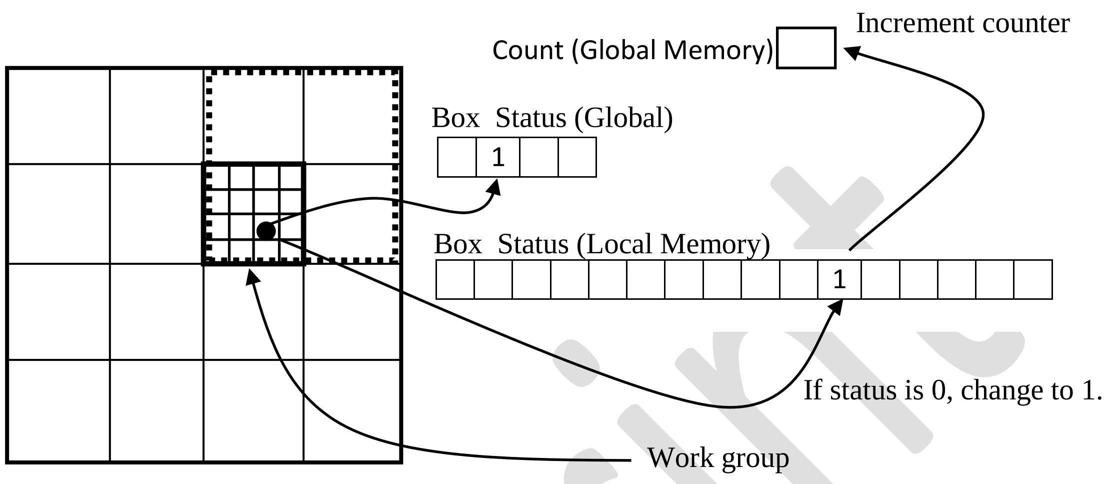


Figure 7. The improved version of the per-pixel algorithm for updating the box status and box count, using work groups and local memory.

The distribution of the boxStatus array into smaller chunks of local memory can therefore be very easily implemented with minimal changes in the kernel code. Additionally, local memory operations are much faster than global memory. However, OpenCL implementation constraints on the maximum two-dimensional work group size could prevent the inclusion of boxes of all sizes within a workgroup. For example, if the maximum work group size is 16×16, then the portion of the boxStatus array of sizes 4, 8, 16 is included in the local memory, while the remaining sizes 32, 64 and 128 are represented in global memory. Thus, a 21 element array is defined in the local memory of each work group. The global memory requirement now reduces to

$$B' = \left(\frac{N}{32}\right)^2 + \left(\frac{N}{64}\right)^2 + \left(\frac{N}{128}\right)^2 = \frac{21N^2}{2^{14}} \qquad (2)$$

The reduction of the boxStatus array residing in global memory, in comparison with the size given earlier in Eq.(1) can be seen as the dotted line in Fig. 4. The percentage of reduction achieved in this case is almost 98.5%. On the other hand, if the work group size is 1024, a local memory of size 85 elements for each work group and a global memory of $5N^2/2^{14}$ elements are required for representing box status. The boxCount array which has only around 6 elements is always defined in the global memory.

The implementation of the modified kernel is shown in Fig. 8. As seen in the code, the initialization of the local memory is carried out using a work group barrier, and the remaining part of the code is similar to the previous method, except that there are now two similar update loops, one for the local memory for box sizes 4, 8, 16 (assuming work group size 256) and the other for the global memory for box sizes 32, 64, 128.

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                               CLK_ADDRESS_CLAMP |
                               CLK_FILTER_NEAREST;
__kernel void fractal_dimn(read_only image2d_t imgIn,
                   __global int* boxStatus_G,
                   __global int* boxCount)
{
   __local int boxStatus_L[21];

   int ix = get_global_id(0);
   int iy = get_global_id(1);
   int wid = get_image_width(imgIn);

   int ilx = get_local_id(0);
   int ily = get_local_id(1);
   int lindx = ily*16 + ilx;

   int iu, iv, offset = 0, index, nboxes, bcindx = 0;
   float4 pixel;

   if(lindx < 21) boxStatus_L[lindx] = 0;
   barrier(CLK_LOCAL_MEM_FENCE);

   pixel = read_imagef(imgIn, sampler, (int2)(ix, iy));
   if(pixel.x < 0.5) return;

   for(int boxsize = 4; boxsize <= 16 ; boxsize <<= 1)
   {
          nboxes = 16 / boxsize;
         iu = (ilx / boxsize);
         iv = (ily / boxsize);
          index = iv*nboxes + iu + offset;
          if(atomic_cmpxchg(&boxStatus_L[index], 0, 1) == 0)
                  atomic_inc(&boxCount[bcindx]);
          offset += (nboxes*nboxes);
          bcindx++;
    }

    offset = 0;
    for(int boxsize = 32; boxsize <= 128 ; boxsize <<= 1)
    {
         nboxes = wid / boxsize;
          iu = (ix / boxsize);
          iv = (iy / boxsize);
         index = iv*nboxes + iu + offset;
          if(atomic_cmpxchg(&boxStatus_G[index], 0, 1) == 0)
                  atomic_inc(&boxCount[bcindx]);
          offset += (nboxes*nboxes);
          bcindx++;
    }

}
```

Figure 8. Kernel code for the box counting algorithm (Implementation 2).

The reduction in the time taken by the box counting algorithm with the above kernel can be seen by comparing the graphs in Fig. 9 with Fig. 6. Table 1 provides a better view in terms of the numerical values for the two processors for large images. On the first processor, the percentage of reduction achieved using the modified kernel described in this section over the computation without the OpenCL kernel is almost 97%.

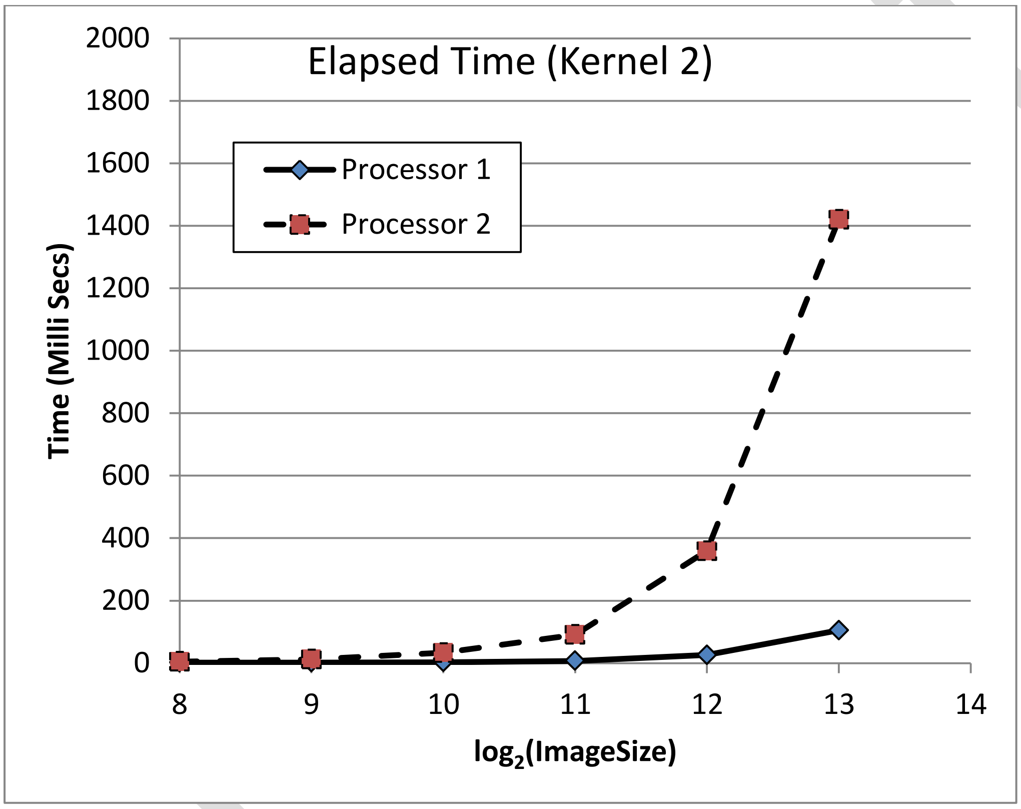


Figure 9. Elapsed time for the execution of the box counting algorithm (Implementation 2) on the GPU of two processors.

	Image Size	Without Kernel	Kernel-1	% Reduction	Kernel-2	% Reduction
Processor-1	2048x2048	180	12	**93.33**	7	**96.11**
	4096x4096	800	43	**94.63**	26	**96.75**
	8192x8192	3700	166	**95.51**	105	**97.16**
Processor-2	2048x2048	270	124	**54.07**	91	**66.30**
	4096x4096	1230	469	**61.87**	359	**70.81**
	8192x8192	5445	1860	**65.84**	1420	**73.92**

Table 1. Comparison of elapsed time on the GPU using two different kernels and two processors

5. Conclusions

The computation of fractal dimension forms an integral part of several texture analysis methods that are commonly used for processing biomedical images. When the image size becomes large, the computation of fractal dimension using the box counting method can take a large amount of time that could have a significant impact on the time taken by the processes of feature extraction and classification. OpenCL provides an ideal framework for implementing the box counting algorithm on the GPU. Moreover, the regular subdivision of an input image into a number of boxes fits very well with the memory model used by OpenCL. This paper has presented implementations that could drastically reduce the computation time for large images. With appropriate distribution of memory within work groups, the amount of global memory required for the process could also be significantly reduced. The paper has presented comparative analysis of the performance of the two kernel implementations on two different processors.

The proposed methods could be easily integrated into existing systems to obtain significant performance gains even on GPUs with a moderately sized number of cores and limited work group sizes. Future work in this area is directed towards improving the kernel design further using OpenCL optimizations, and testing the method on applications requiring the computation of multifractal spectra from several images.

6. Acknowledgement

The work presented in this paper was done as part of a research project funded by the UC College of Engineering Strategic Research Grant.

REFERENCES

1. G. A. Losa, *Fractals in Biology and Medicine* (Wiley Online Library, Oct 2011).
2. J. T. Kuikka, Fractal analysis in medical imaging, *Intl. J.. of Non-linear Sciences and Numerical Simulation*, **3** (2) (May 2011) 81-88.
3. O. Heymans, *et.al.* Is fractal geometry useful in medicine and biomedical sciences?, *Medical Hypotheses*, **54** (3) (Mar 2000) 360-366.
4. C. C. Chen, J. S. DaPonte and M. D. Fox, Fractal feature analysis and classification in medical imaging, *IEEE Trans. on Medical Imaging*, **8** (2) (June 1989) 133-142.
5. P. W. Huang and C. H. Lee, Automatic classification for pathological prostate images based on fractal analysis, *IEEE Trans. on Medical Imaging*, **28** (7) (July 2009) 1037-1050.
6. B. R. Masters, Fractal analysis of the vascular tree in the human retina, *Annual Review of Biomedical Engineering*, **6** (Aug. 2004) 427-452.
7. A. N. Esgiar, R. N. G. Naguib, B. S. Sharif and M. K. Bennett, Fractal analysis in the detection of colonic cancer images, *IEEE Trans. on Information Technology in Biomedicine*, **6** (1) (Mar 2002) 54-58.
8. K. Falconer, *Fractal Geometry – Mathematical Foundations and Applications*, 2nd ed. (Wiley, 2006).
9. M. Schroeder, *Fractals, Chaos, Power Laws*, (Dover, New York, 2009).
10. Y. Shimzu, S. Thurner and K. Ehrenberger, Multifractal spectra as a measure of complexity in human posture, *Fractals*, World Scientific, **3** (1), 183-210.
11. I. Reljin and B. Reljin, Fractal geometry and multifractals in analyzing and processing medical data and images, *Archive of Oncology*, **10** (4) (2002) 283-293.

12. R. Mukundan and Anna Hemsley, Tissue image classification using multifractal spectra, *The Intl J. of Multimedia Data Engineering & Management*, IGI Publishing, **1** (2) (2010) 61-74.

13. R. Lopes and N. Betrouni, Fractal and multifractal analysis: A review, *Medical Image Analysis*, **13** (4) (Aug 2009) 634-649.

14. H. Ahammer, Higuchi dimension of digital images, *PLoS ONE*, **6**(9), e24796.

15. H. Zhang, C. S Perng, and Q. Cai. An improved algorithm for feature selection using fractal dimension, Proc. 2nd International Workshop on Databases, Documents, and Information Fusion (DBFusion 2002), Karlsruhe, Germany, July 4-5 (2002).

16. Tzeng, Y.C., Fan, Y.J., Chen, K.S. A parallel differential box counting algorithm applied to hyperspectral image classifications, *Geoscience and Remote Sensing*, **9 (**2), (2012), 272-276.

17. J. Jiménez and J. Ruiz de Miras. Fast box-counting algorithm on GPU. *Computer Methods and Programs in Biomedicine*, **108** (3) (2012), 1229-1242.

18. A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung and D. Ginsburg, *OpenCL Programming Guide* (Addison-Wesley, 2012).

19. nVidia Developer Zone – OpenCL, https://developer.nvidia.com/opencl (Retrieved Jan 2014).

20. M. Scarpino, *OpenCL in Action*, (Manning, 2012).