Acceleration of the Marching Cubes Technique for Volumetric Visualization on Graphics Processing Unit Using Spatial Data Structures

Marcos Vinícius Mussel Cirne
Institute of Computing
University of Campinas
Campinas-SP, Brazil, 13084-971
marcosvcirne@gmail.com

Hélio Pedrini
Institute of Computing
University of Campinas
Campinas-SP, Brazil, 13084-971
helio@ic.unicamp.br

ABSTRACT

Volume rendering has several applications that benefit many domains of knowledge, such as medicine, biology, geology, and virtual reality. One of the most widely used methods for real-time visualization of data volumes is the marching cubes, which is part of a class of visualization techniques called isosurface extraction. Due to the constant evolution of graphics processing units, it is possible to obtain considerable rendering rates and a high degree of realism. This work describes a methodology for accelerating the marching cubes algorithm on a graphics processing unit and presents some possible ways to improve its performance through auxiliary spatial data structures. Experiments using several volumetric datasets show the effectiveness of the proposed method.

Categories and Subject Descriptors

I.3.1 [**Computer Graphics**]: Hardware Architecture.

General Terms

Graphics Processors, Parallel Processing

Keywords

Marching Cubes, Volume Rendering, Volumetric Data, Graphics Processing Unit, Isosurface Extraction

1. INTRODUCTION

Volume visualization is an approach to information extraction from volumetric data sets and it is related to the representation, manipulation and rendering of such sets. This method emerged during the 1990s as a branch of the scientific visualization field and has applications in various knowledge domains, such as medicine, virtual reality, geology, oceanography, meteorology, among others.

One of the challenges of volume visualization is the development of efficient algorithms, as well as their possible refinements and adaptations. Despite the significant progress in the area, it was observed that the use of a *central processing unit* (CPU) to perform general purpose graphics processing tasks was not enough to achieve better interactivity or real-time rendering, when used in very large data sets. The increasing technological advances enabled the emergence of powerful *graphics processing units* (GPUs), capable of rendering complex three-dimensional models, achieving a high degree of realism.

The potential of GPUs is driven by their high level of parallelism and their ability to perform floating point and geometric primitive operations in a fast and efficient way. Recently, GPUs have been widely used for acceleration of applications (such as image and video processing, fluid dynamics simulation, seismic analysis, cryptography and so on), resulting in a significant gain over the CPUs by an order of magnitude. This has been possible due to the development of the *general-purpose computing on graphics processing units* (GPGPU), making them even more flexible. The idea of this technique is to take advantage of the high parallelism on GPUs and facilitate the application programming in general.

Currently, the most used GPGPU technology is the *Computing Unified Device Architecture* (CUDA), from NVIDIA [19], which allows the use of a C-like programming language for the development of algorithms to run on GPUs. A number of other technologies have also been developed in recent years, as will be shown later.

As a consequence of the evolution of these technologies, volume visualization has also achieved considerable advances. Real-time GPU-accelerated volume rendering became a very effective tool for visualization and analysis of volumetric data.

The purpose of this paper is to describe a volume visualization method, called marching cubes, for isosurface extraction, as well as how it can be optimized by changes in its implementation details and GPU acceleration.

This paper is organized as follows. Section 2 reviews some relevant concepts of volume visualization and isosurface extraction, as well as the marching cubes algorithm and some spatial data structures used to improve the performance of the algorithm. Section 3 describes a methodology for ac-

WebMedia'11: Proceedings of the 17th Brazilian Symposium on Multimedia and the Web. Full Papers.
October 3 -6, 2011, Florianópolis, SC, Brazil.
ISSN 2175-9642.
SBC - Brazilian Computer Society

celerating the marching cubes technique on GPU. Section 4 presents and discusses the experimental results obtained by applying the proposed method to volumetric data. Section 5 concludes the paper with some final remarks.

2. BACKGROUND AND PREVIOUS WORK

This section describes the general background of the volumetric visualization, followed by the concepts of isosurface extraction, marching cubes algorithm, and a description of some data structures used to improve the performance of this algorithm.

2.1 Volumetric Visualization

In general, the volumetric data is usually represented by a set of volume elements, called *voxels*, where each one contains a specific value in a regular grid contained in the three-dimensional space. A voxel can be defined by a tuple $< x, y, z, S >$, which represents the value S associated to some property of a volume data, located at a general 3D position (x, y, z).

According to Elvins [4], the fundamental volumetric visualization algorithms can be classified into two groups: Direct Volume Rendering (DVR) and Surface-Fitting (SF). The first one is characterized by the direct element mapping onto the screen space, without the use of geometric primitives as an intermediary representation, and the second consists of stages of feature extraction and representation of isosurfaces (surfaces that represent a set of points with the same scalar value), which are later rendered for visualization. These isosurfaces can be defined from surface primitives (such as polygons) or by a certain threshold.

Some of the DVR techniques include Raycasting [12, 14], Splatting [22, 24], Cell-Projection [25, 26] and Shear-Warp [11, 13]. Some of the SF techniques are Contour Connection [10, 18] and Marching Cubes [6, 16, 23].

2.2 Isosurface Extraction

An isosurface can be defined as a set of points that have the same value (called *isovalue*) in a volume data, i.e., $\{x \in \Re^3 : f(x) = h\}$, for some isovalue $h \in \Re$. The procedure of isosurface extraction involves the generation of meshes (usually triangular) that approximately represents a certain surface. In the medical field, for instance, this procedure is commonly used in the visualization of organs, tissues and anatomic structures.

The marching cubes algorithm [6, 16, 23] is the main isosurface extraction technique, and it was originally developed to improve the study of 3D medical images. Later, many researches were conducted to optimize this technique through the use of spatial data structures to improve the processing of volume data. However, with the advent of modern graphics cards, techniques that take most advantages from the graphics hardware have been explored due to the high degree of parallelism present in these cards.

2.3 Marching Cubes

The marching cubes technique [6, 16, 23] uses a divide-and-conquer approach in which the volume data is processed through their cells (voxels), that are equivalent to cubes. In each cell, the intersection between its respective edges and the isosurface is verified. The values of each vertex cells are then compared to a given isovalue h and these vertices are classified as "inside" or "outside" the isosurface. The first case is applied when the value of the vertex is greater than or equal to h and the second one when it is less than h. Once defined the type of intersection, an approximation of the isosurface contained in the cell is done by constructing triangles.

As each of the 8 vertex cells has only two possible states, we have then a total of $2^8 = 256$ cases of intersection between isosurface and cell edge, which are listed in a lookup-table. However, some pairs of these cases are symmetric or complementary to each other, which restrains the problem to 15 cases, as shown in Figure 1. A demonstration of the marching cubes algorithm using different isovalues is shown in Figure 2.

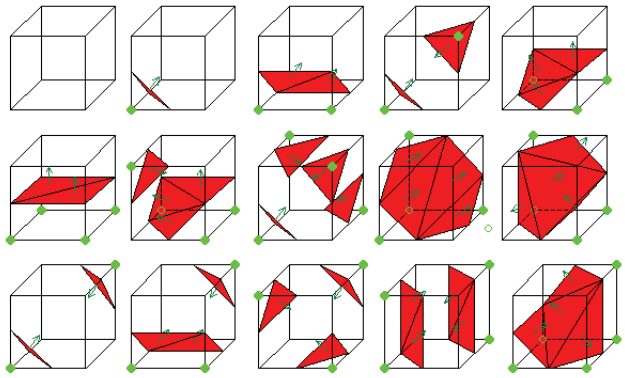


Figure 1: Illustration of the 15 basic cases of the marching cubes technique. The green vertices are the ones classified as "inside" the isosurfaces, and the remaining as "outside" them. Image extracted from [16].

The pseudocode of the algorithm can be briefly described according to the Algorithm 1. The advantage of this algorithm is that the processing of a cell is independent of the other ones, which permits its parallelization. However, as a disadvantage, it may generate holes in the isosurfaces, due to topological ambiguities of the cases.

The algorithm runs in time $O(n)$, where n is the total number of processed cells. On the other hand, many of these cells are empty, incurring an unnecessary waste of time. To minimize this waste, some spatial data structures are used to only process the active cells, i.e., the ones that are intercepted by an isosurface, reducing the complexity of the algorithm.

In the scope of the acceleration of marching cubes algorithm on the GPU, Johansson and Carr [9] conducted a comparative analysis of its execution using data structures, such as *k-d trees* [2] and *interval trees* [3], reporting the rendering rate speedups obtained in relation to the CPU. Newman and Yi [17] developed an in-depth research about the possibili-

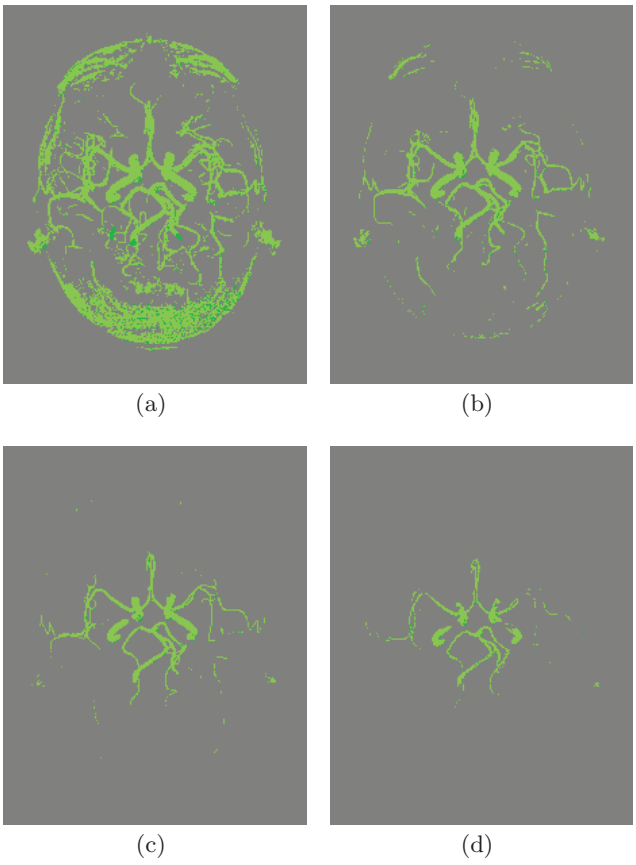


Figure 2: Volume rendering of an angiography dataset using marching cubes with different isovalues: (a) 60; (b) 80; (c) 100; (d) 120.

ties of developing the marching cubes technique, describing their respective properties, extensions and attempts to solve its limitations.

2.4 Accelerating Data Structures

There are several classes of data structures that are very useful to avoid the processing of empty cells. One of them consists of interval-based representations, which uses cell intervals to group cells [17]. The advantage of this type of representation is in its flexibility, being applied not only on regular grids, but also on non-regular grids, once it works from an interval space, instead of using the mesh space itself.

The main methods of this class are based on a representation called *span-space* [15], where each cell of the volume data is mapped to a two-dimensional point, whose coordinates x and y correspond, respectively, to the minimum and maximum values of the cell. From a given isovalue h, the points of the span-space that represent the active cells are the ones where $x \leq h$ and $y \geq h$.

This section describes some spatial data structures and how they can be used to improve the performance of the marching cubes algorithm.

Algorithm 1 Marching Cubes

1: Read four slices from memory.
2: Take two slices and create a cube from 4 neighbors of a slice and 4 of the next one.
3: Calculate an index to the cube, comparing the 8 density values of the cube vertices with the surface constant (isovalue).
4: Using the calculated index, verify the edge list from a lookup-table.
5: Using the densities in each vertex of the edge, find the intersection surface-edge by linear interpolation.
6: Calculate a unitary normal in each cube vertex by the method of central differences [7]. Interpolate the normal to each triangle vertex.
7: Return the triangle vertices and the vertex normals.

2.4.1 *k-d Tree*

The k-d tree [2] is a special case of the binary search tree, used to organize points located in a k-dimensional space. Each non-leaf node represents a splitting hyperplane that divides the space into two parts in a specific direction, which is defined according to the depth of this node in the tree. The left subtree contains all the points located at the left of the hyperplane and the right subtree contains the ones to the right. The leaf nodes store one point each.

In the marching cubes algorithm, the volume data is mapped onto a span-space before constructing the tree, once the queries in the k-d tree are faster when working with points in a 2-D plane rather than in a 3-D space. Furthermore, every node stores a point in the span-space, instead of storing the points only in the leaves. The construction takes $O(n \log n)$ time, where n is the total number of cells in the volume data.

When searching in the tree, given an isovalue h, it will traverse only the nodes that correspond to the active cells of the volume data. Thus, the query takes $O(\sqrt{n} + p)$ time, where p the number of active cells.

2.4.2 *Interval Tree*

The interval tree [3] is an ordered tree used to store intervals of values in 1-D. Similarly to the k-d tree, it is an extension of the binary search tree, and allows an efficient search of all intervals that overlap with a given interval or point.

The root of the tree stores a value that corresponds to the median of the endpoints of all intervals and a list of intervals that contain this value. The left subtree stores the intervals that are completely below the median and the right subtree stores the ones completely above the median. Then, the process is repeated recursively for each subtree. It takes a construction time of $O(n \log n)$.

The span-space is suitable for constructing an interval tree. In this case, the intervals correspond to the volume cells, and the endpoints are the minimum and maximum values of the cell, which stands for its coordinates in the span-space.

Searching in an interval tree takes $O(\log n + p)$ time, which makes it more efficient than the k-d tree. On the other hand,

it demands higher memory space.

2.4.3 Quadtree and Octree

A quadtree [5] is a tree data structure where every non-leaf node has exactly four children. It is used to partition a region in a 2-D space into four equal regions (or quadrants). These regions are then partitioned into other four subregions, and so on, until the subregion is empty, which characterizes a leaf node. The 3-D analogous structure of a quadtree is called octree, which partitions a 3-D space region into eight subregions (or octants).

Once the span-space is a 2-D space, it can be represented by a quadtree. Let l be the number of bits used to store the volume data values, which means that there are 2^l possible values (ranging between 0 and $2^l - 1$) for a vertex cell. Thus, the span-space is a $2^l \times 2^l$ region, and the points correspond to the mapped cells. Every node in the tree stores the information of a point in the span-space.

However, more than one cell can be mapped to a same point in the span-space. To overcome this problem, each node in the quadtree also stores a pointer to a list of the volume data cells that were mapped to this point. Then, the queries can be made as usual, traversing the nodes corresponding to the active cells.

In the octree case, the tree is built directly from the volume data. But in case of non-regular grids (i.e., when volume dimension is not a power of 2), the subregions have different sizes, once we are partitioning cell regions.

3. METHODOLOGY

The significant evolution of the programmability of the graphics hardware allowed the isosurface extraction procedure to be accelerated by the GPU, taking advantage of its parallel architecture. However, the bottleneck of the acceleration is in the communication bus between the CPU and the GPU, which means that not always the best solution is to transfer all the tasks to the GPU. Thus, in order to maximize the performance of this procedure, it is necessary a good planning of the graphics pipeline, selecting the tasks that can be run on the CPU and the ones that can be transferred to the GPU, as well as a proper use of all the memory hierarchy. Some proposals of graphics pipeline for isosurface extraction were made by Pascucci [21] and Johansson [8].

The methodology proposed in this work is restricted to the marching cubes technique, instead of generalizing to the isosurface extraction process. A general scheme is shown in Figure 3, composed of six stages. At the Stage 1, the CPU reads a volume data, which is then allocated both in the main memory (RAM, used by the CPU) and in the video memory (VRAM, used by the GPU). Later, one of the spatial data structures described in Section 2.4 is constructed from the volume data and stored only in the main memory (Stage 2).

Once finished the preprocessing stage, the marching cubes algorithm is started (Stage 3). From an isovalue h specified by the user, the CPU makes a search in the data structure, traversing only the nodes corresponding to the active volume cells, and generating a list of these cells, which is then transferred to the GPU by a communication bus.

With the list of active cells and the isovalue h, the GPU continues the marching cubes algorithm. Each cell is classified into one of the 15 cases of marching cubes (shown in Figure 1) by comparing h to the 8 cell vertices. From this comparison, a cell index is created and then used to determine the number of vertices needed to render the isosurface contained in the cell. After this procedure is done for all active cells, the exact total number of vertices to be output can be determined, which will then define the size of video memory needed to allocate the vertex buffers: one for storing these vertices and other for their respective normals (Stage 4).

After that, the GPU once more traverses the list of active cells to generate the triangles that comprise the isosurfaces (Stage 5). For each active cell of the list, it creates the same cell index as described above, which in GPU is faster than storing the previous results in an array and then retrieving the respective values. Furthermore, it calculates the isosurface intersections with the 12 edges of the cell by interpolating the vertices and the normals calculated by the GPU from the volume data. Later, with the cell index and the intersections, the GPU obtains the list of vertices and normals related to the isosurface, writing them in the respective vertex buffers. Finally, the volume data is rendered from these buffers (Stage 6).

All the procedures executed by the GPU are parallelized, once the results obtained from a cell are independent of the others. However, the acceleration of the marching cubes relies on the way this parallelization occurs. When a task is assigned to the GPU, it creates a specific amount of blocks [1], which contain the same number of threads, responsible for running a part of this task.

In our approach, the amount of blocks depends on the number of active cells and the number of threads per block, also called *block size*. The block size is chosen in such a way that it is neither too low, assigning too much work for all threads and not maximizing the task performance at all, nor too high, causing an overhead of starting and terminating threads.

4. EXPERIMENTAL RESULTS

The tests were executed on an AMD Phenom II X6 1090T 3.2 GHz processor with 4 GB of RAM, and an NVIDIA GeForce GTS 450 with 1 GB of VRAM, using a C-like programming language, OpenGL and CUDA API's.

The experiments were made using 8-bit datasets from [1], where each scalar value ranges from 0 to 255. Table 1 shows a list of volume datasets used in the tests, together with their respective dimensions (in voxels), isovalues (input to the marching cubes algorithm) and number of triangles rendered in the screen (which depends on the isovalue).

Figure 4 shows two plots that illustrate the average time of 50 executions of our marching cubes implementation on the GPU for different values of block size, using each of the vol-

[1]Concept from CUDA Programming Model [19].

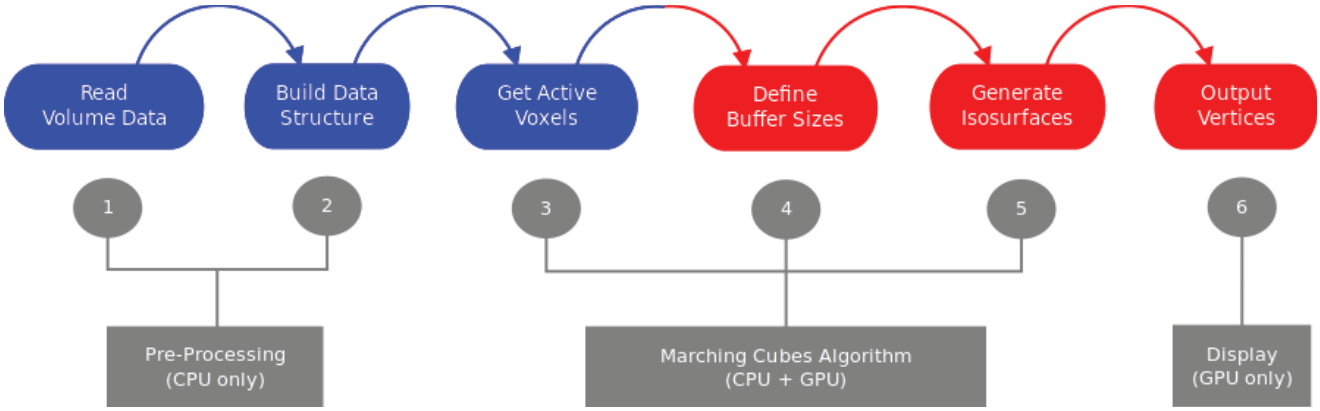


Figure 3: Our approach for acceleration of the marching cubes algorithm. Stages 1 and 2 responsible for the preprocessing stage; stages 3 to 5 correspond to the execution of marching cubes and stage 6 makes the display of vertices on the screen. Blue boxes stand for the actions executed on CPU, and the red boxes the ones on GPU.

Volume Name	Dimensions	Isovalue	# Triangles
Fuel	$64 \times 64 \times 64$	10	11534
Hydrogen Atom	$128 \times 128 \times 128$	20	47864
Angiography	$256 \times 320 \times 128$	80	84974
Ventricles	$256 \times 256 \times 124$	120	167214
Engine	$256 \times 256 \times 128$	155	207592

Table 1: List of volume datasets with their respective sizes, isovalues, and number of triangles.

ume datasets and their respective isovalues listed in Table 1. The plot (a) represents the executions of the brute force version of the marching cubes algorithm, i.e., a naive implementation that runs without the aid of accelerating data structures, and (b) the executions using an interval tree to store the voxels from the datasets. From both plots, it can be noticed that for block sizes of 64 and higher, the running time of the marching cubes algorithm is almost the same, and it does not depend on whether a data structure is used or not for acceleration. As stated in Section 3, although a higher number of threads means higher parallelization, the time spent to create the threads also gets higher, hence stabilizing the performance. Thus, the block size used to run our tests was defined at 64.

Table 2 shows the average framerate of the execution of marching cubes algorithm in CPU and GPU, for all data structures described in Section 2.4, and comparing to the brute force version of the algorithm. These results do not consider the time spent on the volume data reading and the data structure construction (except for the brute force marching cubes), considering only the events that occur between the search in the data structure and the volume display on the screen.

As it is possible to observe, the interval tree provided the best results (highlighted in bold in Table 2), not only among all the data structures used in the tests, but also in the acceleration factor compared to the brute force CPU marching cubes, achieving a speedup of 16.4 times, in case of the angiography dataset. This was expected because the interval tree has, asymptotically, a better query time than the other structures. The acceleration factor between CPU and GPU (for the same data structure) is, on average, between 2.0 and 6.0.

Comparing the k-d tree against the octree, we can state that the latter has better results when running marching cubes in CPU, but it is worse than the former in GPU. This can be explained by the fact that the k-d tree uses the span-space approach, unlike the octree, which works directly on the volume data.

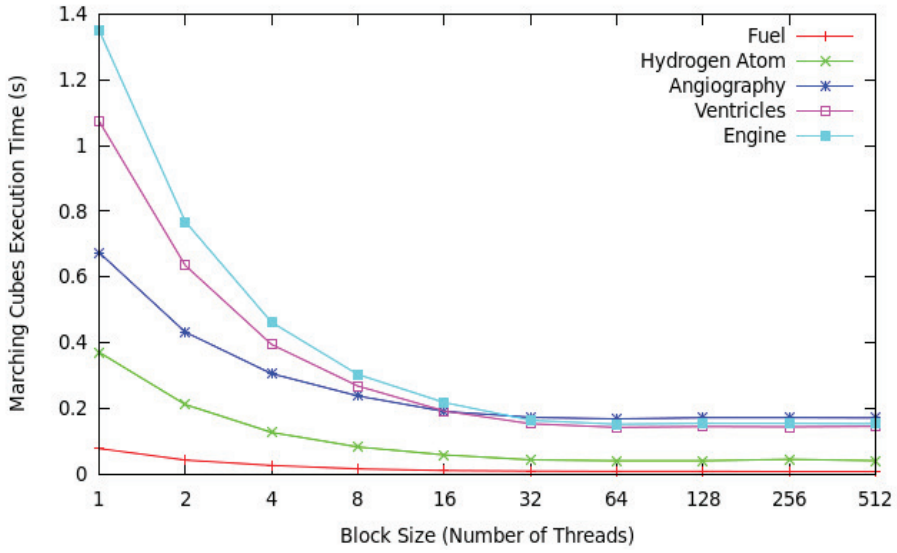
The bottleneck of this approach resides on the fact that the searches in the data structure are done on the CPU. Once tree search algorithms are generally recursive and CUDA does not support recursive algorithms, if non-recursive versions of these algorithms were implemented on the GPU, there would have a large waste of time and space to create stacks and loops.

5. CONCLUSIONS

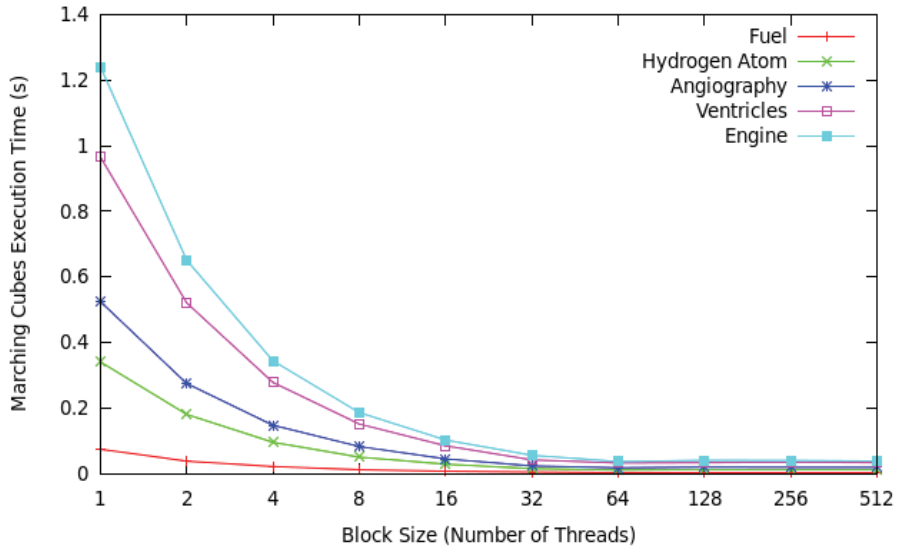
This paper described a method for accelerating volumetric visualization in graphics hardware using the marching cubes algorithm, analyzing its performance when using different spatial data structures, which makes the time complexity of the algorithm be given based on the volume data cells that contain an isosurface, rather than the total number of cells.

Our results demonstrate that it is possible to speed up the algorithm by a factor of approximately 16 times when compared to standard CPU marching cubes.

We plan to expand the proposed method to open frameworks, such as OpenCL [20], once that CUDA framework is restricted to NVIDIA graphic cards. We would also like to make the same analysis shown in this paper with different graphics cards, instead of using a specific card. Finally, we intend to integrate the method into a web-based virtual



(a)



(b)

Figure 4: Average times of executions of the marching cubes algorithm on GPU for different block sizes. Graph (a) shows the results for the brute force version and (b) for the interval tree.

environment for medical training.

6. ACKNOWLEDGMENTS

The authors are grateful to FAPESP, CNPq, and National Institute of Science and Technology in Medicine Assisted by Scientific Computing (INCT/MACC) for the financial support.

7. REFERENCES

[1] Volume Datasets. `http://www.volvis.org`.

[2] J. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] P. Cignoni, P. Marino, C. Montani, and R. Scopigno. Speeding up Isosurface Extraction using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3:158–170, 1997.

[4] T. Elvins. A Survey of Algorithms for Volume Visualization. *SIGGRAPH Computer Graphics*, 26(3):194–201, 1992.

[5] R. Finkel and J. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, Mar. 1974.

[6] F. Gong and X. Zhao. Three-Dimensional Reconstruction of Medical Image Based on Improved Marching Cubes Algorithm. In *International Conference on*

CPU					
Volume Name	**Brute Force**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Fuel	82.4	106.0	**111.5**	107.5	104.7
Hydrogen Atom	14.8	25.5	**32.1**	30.2	28.8
Angiography	3.6	8.2	**11.5**	10.1	9.2
Ventricles	4.0	7.7	**10.8**	10.1	9.4
Engine	3.7	6.4	**8.9**	8.4	7.7
GPU					
Volume Name	**Brute Force**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Fuel	94.6	149.5	**178.9**	170.4	146.9
Hydrogen Atom	25.3	67.6	**88.5**	87.3	49.7
Angiography	6.1	39.2	**59.0**	51.0	28.2
Ventricles	6.8	20.7	**32.4**	27.5	19.7
Engine	6.4	17.5	**28.5**	22.1	16.9

Table 2: Average frame rate (in frames per second) of the execution of marching cubes algorithm in CPU and GPU, with different data structures, comparing to the brute force version.

Machine Vision and Human-Machine Interface, pages 608–611, Kaifeng, China, Apr. 2010.

[7] H. Jeffreys and B. Jeffreys. Central Differences Formula. In *Methods of Mathematical Physics*, pages 284–286. Cambridge University Press, Cambridge, England, 1988.

[8] G. Johansson. Accelerating Isosurface Extraction by Caching Cell Topology with Graphics Hardware. Master's thesis, University College Dublin, Dublin, Ireland, 2005.

[9] G. Johansson and H. Carr. Accelerating Marching Cubes with Graphics Hardware. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, Toronto, ON, Canada, 2006.

[10] E. Keppel. Approximating Complex Surfaces by Triangulation of Contour Lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[11] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, Stanford, CA, USA, 1996.

[12] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[13] T. Li, M. Xie, W. Zhao, and Y. Wei. Shear-Warp Rendering Algorithm Based on Radial Basis Functions Interpolation. In *Second International Conference on Computer Modeling and Simulation*, pages 425–429, Jan. 2010.

[14] B. Liu, G. Clapworthy, and F. Dong. Accelerating Volume Raycasting using Proxy Spheres. *Computer Graphics Forum*, 28(3):839–846, June 2009.

[15] Y. Livnat, H.-W. Shen, and R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, Mar. 1996.

[16] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[17] T. Newman and H. Yi. A Survey of the Marching Cubes Algorithm. *Computers & Graphics*, 30(5):854–879, Oct. 2006.

[18] K. Nurzyńska. 3D Object Reconstruction from Parallel Cross-Sections. In L. Bolc, J. Kulikowski, and K. Wojciechowski, editors, *Computer Vision and Graphics*, volume 5337 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2009.

[19] NVIDIA CUDA C Programming Guide Version 3.2. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[20] OpenCL. The Khronos Group – `http://www.khronos.org/opencl/`.

[21] V. Pascucci. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Proceedings of Joint Eurographics - IEEE TVCG Symposium on Visualization*, pages 293–300, Konstanz, Germany, May 2004.

[22] P. Schlegel and R. Pajarola. Layered Volume Splatting. In *Advances in Visual Computing*, volume 5876 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2009.

[23] Z. Wang, B. Fan, N. Li, and H. Zhang. Iso-surface Extraction and Optimization Method Based on Marching Cubes. In *International Conference on Semantics, Knowledge and Grid*, pages 458–460, Zhuhai, China, Oct. 2009.

[24] L. Westover. Footprint Evaluation for Volume Rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 367–376, Dallas, TX, USA, 1990.

[25] J. Wilhelms. A Coherent Projection Approach for Direct Volume Rendering. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1990.

[26] S. Zhu and Y.-L. Gu. Volume Rendering Algorithm of Irregular Volume based on Cell Projection. *Computer Engineering and Applications*, 44(15):68–70, May 2008.