Fast CUDA-Based Triangulation of Molecular Surfaces

Sérgio Dias Abel Gomes
Instituto de Telecomunicações,
Departamento de Informática, Universidade da Beira Interior,
Rua Marquês D'Ávila e Bolama,
6200−001 Covilhã, Portugal
sergioduartedias@sapo.pt, agomes@di.ubi.pt

Abstract

Modeling molecular surfaces enables us to extract useful information about interactions with other molecules and measurements of areas and volumes. Over the years many types of algorithms have been developed to represent and rendering molecular surfaces, but all these algorithms have problems related to time performance in triangulating molecular surfaces. One possible solution to solve this problem is using parallel computing systems, but until recently they have been very expensive. Fortunately, the appearance of the new generation of low-cost GPUs with massive computational power opens up an opportunity window to solve this problem. So, in this paper, we present a GPU-based algorithm to speed up the triangulation and rendering of molecular surfaces. Besides we carry out a study that compares a sequential version (CPU) and a parallel version (GPU) of a molecular surface representation using the Marching Cubes (MC) Algorithm.

Keywords

Connolly surface, molecular surface, implicit surface, GPU computing.

1 INTRODUCTION

The study of molecular surfaces was initiated by Richards in 1974 to solve the protein folding problem that consists in predicting the three dimensional structure of a protein [Agostino 08]. This problem happens because at that time there was not a method to represent a 3 dimensional structure of a molecular surface. Consequently, it was not possible to know other aspects of a molecule, namely substrate binding, catalysis and drug-nucleid acid interaction, which require a formal geometrical representation.

To solve this problem, Lee and Richards proposed a new representation for molecular surfaces [Lee 71]. They formulated the molecular surface from the van der Walls surface (also called VDW surface), which is simply the surface of the union of the atoms of a molecule. The Lee-Richards surface is a kind of inflated or offset VDW surface, where the offset displacement is given by the radius of the probe sphere (representing the solvent or water molecule) that abstractly rolls on the molecule atoms. Similar to VDW surfaces, Lee-Richards surface also present crevices and other singularities that originate difficulties in shape matching, as needed in docking.

To solve this latter problem, i.e. to remove the singularities mentioned above, and get a smooth surface, Connolly implemented a new molecular surface model given by a network of two types of surface patches [Connolly 83]:

- *Contact Surface Patches*. These patches concern the surface patches of the atoms that are accessible to or in contact with the probe sphere. Thus, these patches are convex.
- *Re-entrant Surface Patches*. These patches concern the surface patches generated from the surface of the probe sphere when it rolls on the atoms. Some of these patches are concave, some are saddle toroidal.

This new model of molecular surface was given the name of solvent-excluded surface (SES or Connolly surface) [Sanner 98]. The result is a smooth molecular surface given by stitching together convex, concave, and toroidal surface patches. This stitching operation is not that easy because it requires the computation of the curves of contact between neighbour patches, as well as to find the differential conditions that guarantee the smoothness between two neighbour patches.

Alternative algorithms have been used to represent molecular surfaces such as, for example, Ray Tracing [Blinn 82] and Marching Cubes [Lorensen 87]. Marching Cubes requires a kind of isosurface, not a surface formulated as being compounded from patches. The MCs algorithm generates a triangular mesh that approximates the isosurface. Unfortunately, this algorithm is very time-consuming for a large number of voxels. To overcome this problem, researchers have used parallel systems to deal with large vox-

elizations, but they are very expensive. But now, we can take advantage of the low-cost programmable GPU graphics cards to render surfaces of molecules with thousands of atoms.

So, the purpose of this paper is just to describe a new GPU-specific algorithm to triangulate and render molecular surfaces that implements a CUDA-based parallel version of Marching Cubes. Besides, we carry out a comparative study of the time performance between CPU- and GPU-based triangulation algorithms to render molecular surfaces.

This paper is organised as follows. Section 2 describes the mathematical formulation of Connolly surfaces. Section 3 briefly introduces the CUDA technology. Section 4 describes the CUDA-based triangulation algorithm for Connolly surfaces. Section 5 carries out a time performance analysis and comparison between CPU- and GPU-based marching cubes algorithms. Finally, Section 6 summarises the main results obtained with our algorithms, and points out directions for future work.

2 ANALYTIC MOLECULAR SURFACES

A molecule as a set of atoms can be mathematically described as a union of balls, each representing an atom. In this paper, we are interested in analytical formulations of molecular surfaces that result from summing up local functions that describe the electrical field of atoms. This is so because each atom has its own electrical field, which can be described by an analytical function that decays with the distance. It is clear that there several functions that can describe the electrical field of an atom, namely: Gaussian function [Blinn 82], Wyvill function [Wyvill 86], and inverse quadratic distance function. In this paper, we use the inverse quadratic distance kernel given by:

$$f_i(x,y,z) = \frac{C}{r_i} \tag{1}$$

where C stands for the smoothness or blobiness parameter and $r_i = (x-x_i)^2+(y-y_i)^2+(z-z_i)^2$ is the squared distance from the center (x_i, y_i, z_i) of the atom i to a generic point (x, y, z). Typically, the parameter C takes on a value in the interval $]0, 1]$. Thus, the electrical field of each atom in the molecule is formulated as a distance function, *i.e.* an implicit scalar function. The molecular surface is the result of the summation of the distance functions associated to all atoms, *i.e.* the summation of electric fields of all the atoms, as follows:

$$F(x,y,z) = \sum_{i=0}^{N} f_i(x,y,z) \tag{2}$$

where N is the total number of atoms of the molecule. The atomic local functions f_i work as blending functions of the resulting molecular surface, which will be hopefully smooth. The smoothness of the molecular surface depends on the smoothness of the blending functions. Thus, the blending functions must be differentiable. A surface with this characteristics is referred to as *convolution surface* [Bloomenthal 91].

A molecular surface formulated in this manner is an *implicit surface* that is defined by a function F with domain $\mathbb{R}^3$ and range $\mathbb{R}$. This implicit molecular surface is given by all the points in $\mathbb{R}^3$ that satisfy the function $F(x, y, z) = T$, where T is the isovalue. That is, it corresponds to the level set defined by the constant T. Thus, $F > 0$ for points outside the surface, and $F < 0$ for points inside the surface, and $F = T$ for points on the surface.

3 GPU PROGRAMMING USING CUDA

We took the decision of running our program on GPU because our algorithm fits very well in the parallel computing model. In fact, the voxelization of the axis-aligned bounding box that encloses the molecule allows us to speed up the program by assigning a thread to each voxel. So, before proceeding any further, let us first see important details related to CUDA technology and GPU programming.

3.1 CUDA Architecture

CUDA (Compute Unified Device Architecture) gives data-intensive applications access to the tremendous processing power of NVIDIA graphics processing units (GPUs) through a revolutionary computing architecture unleashing entirely new capabilities [Owens 08]. In our work, we have used a NVIDIA GeForce GTX 280 graphics card. This graphics card includes 10 thread processing clusters (TPCs), where each TPC consists of 3 streaming multiprocessors (SMs), with each SM having 8 stream processors, which yields 240 cores in total. Each core processes threads; hence the parallel computing model on GPU. To control and distribute the massive computing load by the 240 cores, we use a thread scheduler to guarantee a nearly full utilisation of the GPU [NVIDIA 08b].

3.2 CUDA Programming Model

When we are programming in CUDA we must take care of how a program is going to be executed. In this paper, a CUDA program is written in C, in which we have CPU-specific functions and GPU-specific functions. A CPU-specific function runs on the CPU (host), while a GPU-specific function is executed on the GPU (device) as shown in Figure 1. There are three types of GPU-specific qualifiers. The first, named `__global__`, acts as prefix of functions that are GPU kernels, that is GPU functions that are invoked from CPU-specific code. The second, named `__device__`, qualifies a GPU function that can be only called from a GPU kernel. Finally, the third qualifier is the keyword `__shared__` that acts as a prefix of a variable allocated in the streaming multiprocessors shared memory. Note that GPU-specific functions are prohibited to be recursive.

There are other three important CUDA functions to (dis)allocate memory on the GPU side and to exchange data between CPU and GPU. The `cudaMalloc` is used to allocate memory on GPU, while the `cudaFree` function releases memory space from GPU. The `cudaMemcpy`

function serves to transfer data from the CPU side to GPU side, and vice-versa. Note that GPU runs kernels. A kernel is a GPU-specific function that launches threads, hierarchically organised into grid–blocks–threads [NVIDIA 08a, Owens 08], as illustrated in Figure 1.

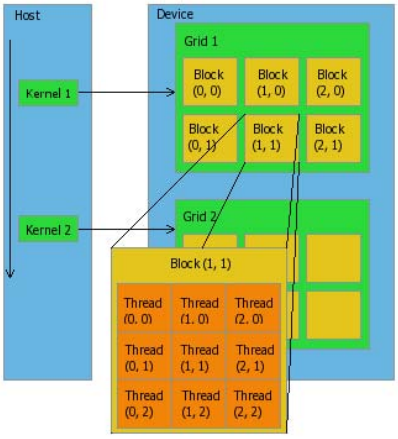

Figure 1. GPU Architecture [NVIDIA 08b]

The compilation of a CUDA program is done in three stages. The first extracts the CPU code from the CUDA file into an intermediate file that is then passed to the standard C compiler. Next, the remaining code, that is the GPU code, is converted to a PTX file (i.e., a kind of assembly language). The third stage translates this PTX file into GPU-specific commands and encapsulates them in an executable file.

4 TRIANGULATION OF CONNOLLY SURFACES

Amongst all the available triangulation algorithms used to render molecular surfaces, the Marching Cubes algorithm is possibly the most adequate for GPU processing. This is justified by the fact that MCs lead to an uniform space partition of the bounding box into voxels, which is ideal for the design of a possible parallel implementation on GPU.

4.1 Marching Cubes: Overview

This algorithm starts by dividing the space that contains the molecule into an uniform grid of voxels of appropriate size. The next step is the calculation of the electrical field intensity F at all voxel vertices of the grid (Equation 2). Doing this involves very time-consuming computations, due to the fact that organic molecules may have long chains of atoms, and the spatial distribution of these atoms often requires very large grids, even for molecules with a small number of atoms. To deal with this problem, our algorithm calculates the intensity in a per atom charge fashion, instead of calculating it at each grid vertex directly. Since the influence of the field can be neglected beyond some appropriate distance, this optimisation can be used in general for intensity calculations. Next, we calculate the position of the voxel for each atom, and then we define a sub-grid centered on such a atom/voxel. The intensity contribution of this atom adds to the current intensity of each sub-grid vertex.

Terminated the computation of the intensity F for all grid vertices, we have to determine the 8-bit flag for each voxel. This flag characterises the surface inside each voxel. The flag bits form an index to a lookup table to determine which edges of each voxel intersect the isosurface. If $F \geq T$ at a grid vertex, the corresponding flag bit is set to 1; otherwise it is set to 0. After, setting the flag bits for each voxel, the algorithm uses the lookup table to determine which of the 256 cases of surface fits inside the voxel. When this is done, the algorithm goes a step forward to interpolate the surface vertices, for each cube along the appropriate edges, by using the lookup tables. After computing the surface vertices for all voxels, we are ready to generate the triangles inside each voxel, which together form the the final mesh that approximates the molecular surface. It is clear that rendering this surface mesh requires the calculation of the triangle normals. The reader is referred to Lorensen and Cline [Lorensen 87] for further details on MCs.

4.2 GPU Implementation

We use CUDA to implement a variant of Marching Cubes (MC) algorithm for molecular surfaces. All MC computations are carried out on the GPU. The algorithm can be described as follows:

- *Bounding Box Computation* (CPU side). We first determine the axis-aligned bounding box that encloses a given molecule is determined while reading in the centers of atoms from a PDB file into an 1-dimensional array of size M, where M is the number of atoms of the molecule. The computation of the bounding box means here computing the locations of the opposite vertices of the diagonal of the bounding box.

- *Voxel Decomposition* (CPU side). Next, we determine the number $N = I \times J \times K$ of cubic voxels with a predefined size, where I, J, and K stand for the number of voxels along x-, y-, and z-axis, respectively, that fit the bounding box.

- *Allocation and Copying of Data Structures to GPU*. This step involves the allocation of GPU memory for two 1-dimensional arrays, to where we copy the array of M atomic centers and the array of N integers, one integer for each voxel; these two operations are carried out by calling the `cudaMallloc` and `cudaMemcpy` functions from the CPU side. These two functions are also used to allocate and copy the lookup tables of the MC algorithm, as well as the array of all voxel vertices, to GPU memory. The previous array of N integers serves the purpose of storing the number of vertices of the triangulation of the molecular surface patch inside each voxel, as needed in a later step.

- *Computation of Electric Field Intensities at Voxel Vertices* (1st GPU kernel). This kernel computes the intensity of electric field at each voxel vertex; this value is initialised to zero at each vertex. Because the number of voxels largely exceeds the number of atoms, the intensity computations are carried out per atom rather than per voxel. This is reinforced by the fact that many biological molecules have long chains of carbon atoms which are not attached to the chains, which implies that the spatial distribution of these atoms originates very large bounding boxes even for molecules with a small number of atoms. The per-atom intensity computation only occurs within a boxed neighbourhood around each atom, *i.e.* at the vertices of voxels surrounding each atom center. The local box centered at each atom center has usually $20 \times 20 \times 20$ voxels. Thus, we are assuming that the local function f_i (see Eq. 1) of the atom i beyond its surrounding box takes on the value 0. In short, the intensity contribution f_i of the atom i at each vertex of its surrounding box is added to the current intensity value F of such vertex.

- *Computation of MC Configurations for Voxels* (2nd GPU kernel). After computing the intensities F at the grid of vertices, we determine the 8-bit flag —a bit per voxel vertex— that corresponds to 1 out of 256 MC surface patterns we may have inside each voxel. A bit has the value 0 if the corresponding vertex has an intensity under the pre-defined isovalue, and 1 if the intensity is over the isovalue. This 8-bit flag works as an index to the lookup table for determining which edges of the voxel intersect with molecular isosurface. This gives the number of vertices of the mesh that approximates the molecular surface inside each voxel; this number is stored into the array of N numbers mentioned above (3rd step). Each element of this array stores the number of vertices we use to triangulate the surface inside each voxel.

- *Allocation of VBO Array for Posterior Rendering of Surface Mesh* (3rd GPU kernel). By performing a Parallel Prefix Sum (scan) [Harris 07] of the number of mesh vertices in the array of N numbers of the previous step, we determine the total number of vertices that sample the molecular surface, from which we triangulate the surface. Recall that this triangulation will be done locally inside each voxel, as usual in the MC algorithm. This Parallel Prefix Sum operation is carried by calling the function `cudppScan`, which is essentially another GPU kernel. The number of vertices output by `cudppScan` is then used to allocate a VBO (Vertex Buffer Objects) array for the mesh vertices, from which we can later generate triangles to render the mesh that approximates the molecular surface.

- *Setting up a Mapping Array Between Array of N Voxel Numbers and VBO Array* (4th GPU kernel). This kernel creates an intermediate array of N elements between the array of N voxel numbers and the corresponding VBO array, which allows to map a given voxel (and its surface vertices) to its first surface vertex in the VBO array. That is, for each voxel, the mapping array stores the index of the VBO array element where we will later store the first surface vertex associated with such a voxel. This allows to create an association between the occupied voxels (*e.g. voxels transverse to the molecular surface*) and the correct position in the VBO array.

- *Computation of Surface Vertices by Linear Interpolation* (5th GPU kernel). This GPU kernel uses linear interpolation along the voxel edges that intersect the molecular surface in order to sample the surface. The lookup tables are here useful to tell us which are the edges that intersect the surface. The resulting sampling points of the surface will be the triangle vertices of the mesh that approximates the surface. These surface points are computed for each voxel and are stored into the VBO array using the previous mapping array.

- *Computation of Surface Normals at the Vertices in VBO Array* (6th GPU kernel). Taking into account that the vertices stored in the VBO array are points of the molecular surface, the normal at a given surface point is given by the gradient vector as follows:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right) \tag{3}$$

where

$$\frac{\partial F}{\partial x} = \sum_{i=1}^{N} \frac{-2\,C\,(x - x_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

$$\frac{\partial F}{\partial y} = \sum_{i=1}^{N} \frac{-2\,C\,(y - y_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

and

$$\frac{\partial F}{\partial z} = \sum_{i=1}^{N} \frac{-2\,C\,(z - z_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

These normals are stored in a separate array.

- *Rendering the Molecular Surface* (CPU side). Taking into consideration that the VBO array in the GPU can be accessed from the CPU side, we have only to display the VBOs to render the surface. We use Gouraud shading that comes with OpenGL.

5 RESULTS

In our tests, we have used a Windows XP PC equipped with a Quad Core Q9550 running at 2.83 GHz, and 4GB RAM, and a NVIDIA GeForce GTX 280 CUDA-programmable graphics card. We have written two programs for the same algorithm:

- *CPU-based Serial Program*. This program was written in C++, and makes usage of the STL (Standard Template Library).
- *CUDA-based Parallel Program*. This program was written in C, and makes usage of the CUDA API.

We have used 16 different molecules, read in from the corresponding .pdb files, to compare the time performance of both algorithms, as shown in Tables 1 and 2. These .pdb files are ASCII files and were taken from the Protein Data Bank (PDB) at `www.rcsb.org`, which is a repository for 3D structural data of molecules. Each molecule has a unique ID (see first column of both Tables 1 and 2).

ID	# Atoms	# Voxels	# Triangles	CPU Time
110D	120	346580	6447	0,43
200D	259	603520	13460	1,2
1QL1	322	1581370	18558	1,9
4PTI	381	1043464	24566	3
1BK2	468	650160	24511	3
2QZF	479	3536664	27386	3,6
2QZD	507	3860800	29642	4
2OT5	545	1154032	32131	4,14
1HH0	691	1632000	25639	4,46
1HGV	691	1970752	25938	4,57
1HGZ	691	1605760	24968	4,36
2INS	781	1316250	43250	7,3
1QL2	966	4619920	56025	11,5
1IZH	1521	2578680	88512	25,1
1GT0	2746	5681590	140821	67,87
1BIJ	4387	7229120	243101	179,1

Table 1. Time performance using CPU.

ID	# Atoms	# Voxels	# Triangles	GPU Time
110D	120	352000	6492	0,13
200D	259	610176	13522	0,26
1QL1	322	1600896	18694	0,39
4PTI	381	1053440	24168	0,4
1BK2	468	657920	24204	0,35
2QZF	479	3562624	27002	0,72
2QZD	507	3901824	29204	0,94
2OT5	545	1167360	32048	0,51
1HH0	691	1652608	25839	0,67
1HGV	691	1994752	25936	0,7
1HGZ	691	1626112	25149	0,64
2INS	781	1326976	42722	0,82
1QL2	966	4654208	56388	2
1IZH	1521	2601472	86380	2,15
1GT0	2746	5720832	138354	6,9
1BIJ	4387	7267456	239896	14,26

Table 2. Time performance using GPU.

The time performance results output by both programs are presented in Tables 1 and 2, where we find columns for the

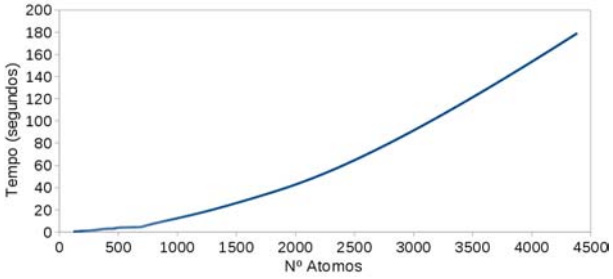

Figure 2. CPU runtime analysis

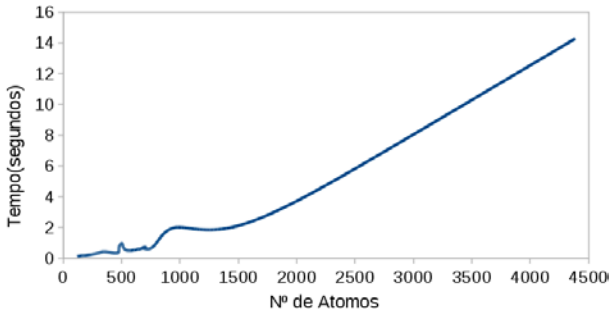

Figure 3. GPU runtime analysis

number of atoms (# Atoms), number of voxels (# Voxels) and number of triangles (# Triangles). Table 1 also has the column 'CPU time' that shows the time performance of the serial program in seconds, while the column 'GPU time' in Table 2 presents the time performance of the CUDA-based parallel program.

Comparing the results obtained in CPU (Table 1) and GPU (Table 2) we note slightly differences in the number of voxels (and, consequently, triangles). These differences are due to the distinct formulas we use to calculate the number of voxels in CPU and GPU. We use the formula $(Bx \times By \times Bz)$ for the CPU, and the formula $(Bx \times By \times Bz) + (By \times Bz) + (Bz)$ for the GPU, where Bx, By, Bz are the sizes of the bounding box along each axis. The rationale behind these different formulas are:

- *Division of GPU Threads*. The number of threads per block —in a grid of thread block— is a multiple of 2; we have used 128 threads per block. This means that the number of voxels of the bounding box that encloses the molecule must also be a multiple of two because each voxel is processed by a single thread.
- *Voxel Indexing*. Taking into account that GPU 3D arrays were not available at the starting time of this project, we had to arrange the indexing formula given above for an equivalent 1D array in the GPU memory.

As expected, and looking at Tables 1 and 2, the main difference between CPU- and GPU-based implementations occurs in time performance, as a consequence of the number of voxels that are processed at the same time. CPU-based implementation processes a single voxel at a time, while GPU-based implementation processes voxels simul-

taneously, with each voxel being processed by a single thread. In true, our GPU implementation was designed in a manner that a thread processes a small number of voxels in sequence. This happens as an consequence of the massive computation that the GPU can process in parallel with its multiple arithmetic logic units. Consequently, GPU-based implementation is much faster CPU-based counterpart.

Figures 2 and 3 show the time complexity of the CPU- and GPU-based algorithms, respectively. Figure 2 shows that the CPU-based algorithm has quadratic complexity because the graph approximates a parabola. On the other hand, Figure 3 shows that the GPU-based algorithm has linear complexity because, apart the unstable behaviour for small molecules (as a consequence of the non-uniform distribution of atoms), the graph approximates a line. This happens because CPU processes a voxel at a time, while GPU processes several voxels simultaneously.

6 CONCLUSION

In this paper we have presented a CUDA-based method to render molecular surfaces on GPU. The main contribution of this work is a multi-threaded GPU-based implementation of Marching Cubes algorithm to triangulate and render molecular surfaces. Also, a comparison to a CPU-based implementation of the Marching Cubes has been carried out for a number of different molecules.

Besides, taking into consideration the linear time complexity of the GPU-based algorithm, we hope in the near future to design and implement a scalable parallel algorithm using a cluster of GPUs in order to process and render large molecules (i.e. with hundreds of thousands of atoms) in real-time.

References

[Agostino 08] Daniele Agostino, Andrea Clematis, Ivan Merelli, Luciano Milanesi, and Matteo Coloberti. A grid service based parallel molecular surface reconstruction system. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 455–462, Washington, DC, USA, 2008. IEEE Computer Society.

[Blinn 82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, July 1982.

[Bloomenthal 91] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *Computer Graphics*, 25(4):251–256, 1991.

[Connolly 83] Michael Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709–713, August 1983.

[Harris 07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum *scan* with cuda. In GPU Gems 3, editor, *Hubert Nguyen*. Addison-Wesley Professional, Upper Saddle River, New Jersey, Aug 2007.

[Lee 71] B. Lee and F. Richards. The interpretation of protein structures: Estimation of static accessibility. *Journal of Molecular Biology*, 55(3):379–380, February 1971.

[Lorensen 87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 163–169, New York, NY, USA, July 1987. ACM Press.

[NVIDIA 08a] NVIDIA. CUDA programming guide 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`, Jul. 2008.

[NVIDIA 08b] NVIDIA. Technical brief information. `http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf`, Jun. 2008.

[Owens 08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[Sanner 98] Michel F. Sanner, Arthur J. Olson, and Jean-Claude Spehner. Reduced surface: An efficient way to compute molecular surfaces. *Biopolymers*, 38(3):305–320, December 1998.

[Wyvill 86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, February 1986.

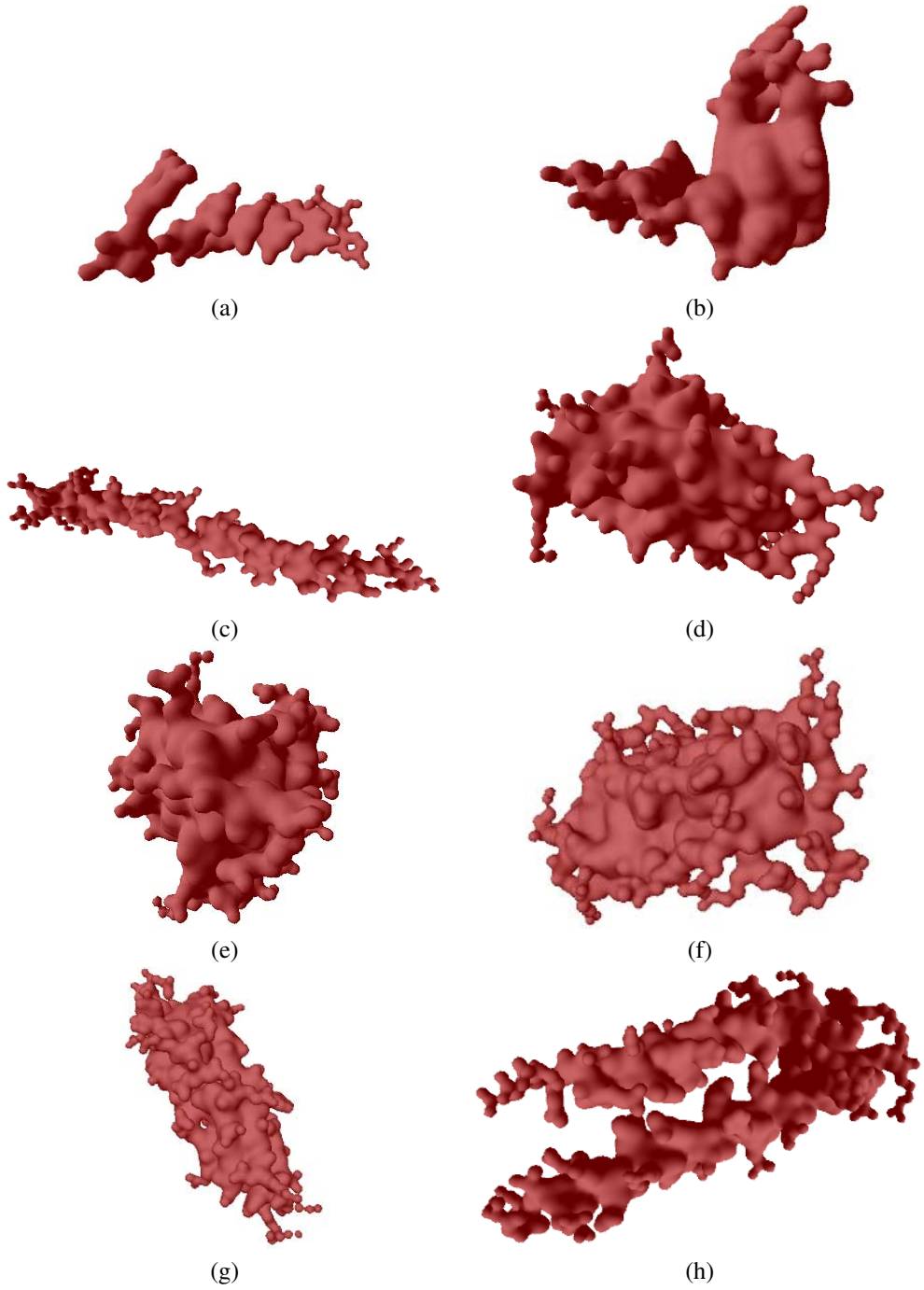

Figure 4. Examples of molecular surfaces displayed using multi-threaded Marching Cubes algorithm. (a - PDB ID:110d), (b - PDB ID:200d), (c - PDB ID:1ql1), (d - PDB ID:4pti), (e - PDB ID:1bk2), (f - PDB ID:2qzf), (g - PDB ID:2qzd), (h - PDB ID:2ot5)

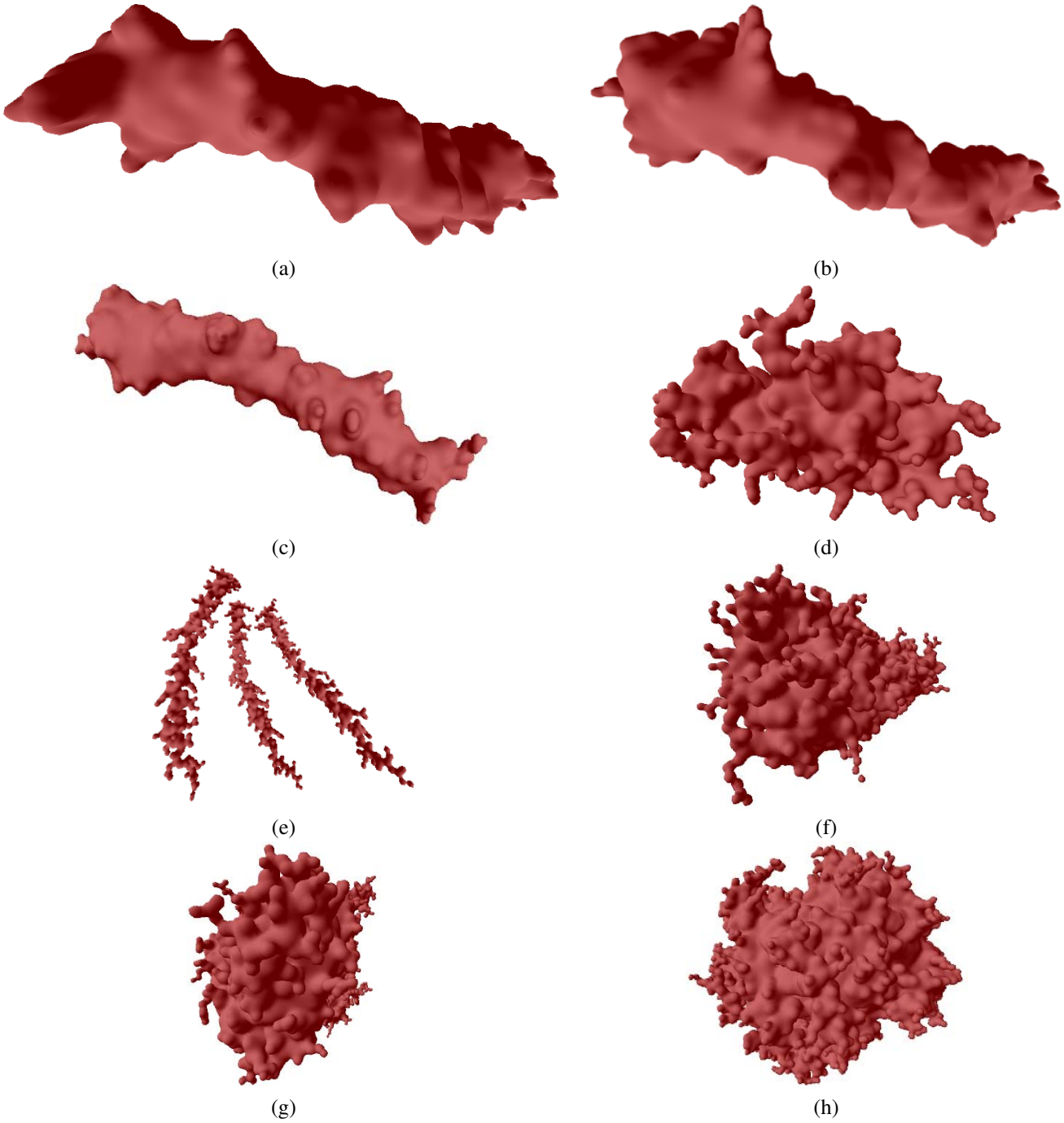

Figure 5. More examples of molecular surfaces displayed using multi-threaded Marching Cubes algorithm. (a - PDB ID:1hh0), (b - PDB ID:1hgv), (c - PDB ID:1hgz), (d - PDB ID:2ins), (e - PDB ID:1ql2), (f - PDB ID:1izh), (g - PDB ID:1gt0), (h - PDB ID:1bij)