Workshop on Molecular Graphics and Visual Analysis of Molecular Data (MolVA) (2019)
J. Byška, M. Krone, and B. Sommer (Editors)

A Massively Parallel CUDA Algorithm to Compute and Visualize the Solvent Excluded Surface for Dynamic Molecular Data

Marco Schäfer[1] and Michael Krone[1]

[1]Big Data Visual Analytics, University of Tübingen, Germany

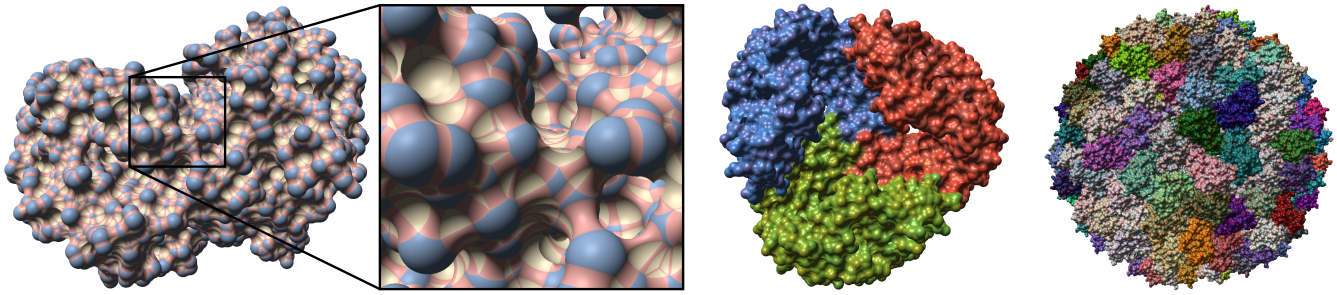

Figure 1: *Screenshots of our Solvent Excluded Surface (SES) visualization method showing three different test data sets obtained from the PDB [BWF*00]. Left: haptoglobin-hemoglobin complex (PDB ID:* 4XOL*) colored by SES patch type (blue: convex spherical patches, red: toroidal patches, yellow: concave spherical triangles). The cutout to the right shows that our method handles all singularities correctly and creates a pixel-perfect image. Middle: an aquaporin (PDB ID:* 1AF6*) colored to show its three different amino acid chains. Right: Cowpea chlorotic mottle virus capsid (CCMV, PDB ID:* 1CWP*) consisting of 214 k atoms, represented by the largest data set used for performance measurements. The probe radius was set to 1.4 Å for all data sets, which approximates water.*

Abstract

The interactive visualization of molecular surfaces can help users to understand the dynamic behavior of proteins in molecular dynamics simulations. These simulations play an important role in biochemical and pharmaceutical research, e.g. in drug design. The efficient calculation of molecular surfaces in a fast and memory-saving way is a challenging task. For example, to gain a detailed understanding of complex diseases like Alzheimer, conformational changes and spatial interactions between molecules have to be investigated. Molecular surfaces, such as Solvent Excluded Surfaces (SES), are instrumental for identifying structures such as tunnels or cavities that critically influence transport processes and docking events, which might induce enzymatic reactions. Therefore, we developed a highly parallelized algorithm that exploits the massive computing power of modern graphics hardware. Our analytical algorithm is suitable for the real-time computation of dynamic SES based on many time steps, as it runs interactively on a single consumer GPU for more than 20 k atoms.

CCS Concepts

• ***Human-centered computing*** → ***Visualization;*** • ***Computing methodologies*** → *Parallel algorithms;* • ***Applied computing*** → *Molecular structural biology;*

1. Introduction

High-quality visualizations of smooth molecular surfaces are an important tool for the analysis of molecular data. The investigation of interactions between large biomolecules—such as proteins—and solvents, ligands, or other proteins, plays an important role in biochemical, medical, and pharmaceutical research. To reach a comprehensive understanding of the diverse and complex molecular interactions requires analytically correct and interactive visualizations. Such interactions occur for example in biochemical pathways as docking events that induce transportation, assembly, or enzymatic reactions. Visualization can help to derive new knowledge for exploring novel drug targets, improving drug design, or identifying potential reaction partners. The shape of a protein can influence these interactions. Thus, visualizations of the molecular surface in

DOI: 10.2312/molva.20191094

combination with a coloring that illustrates the physico-chemical properties influencing these interactions can greatly assist the understanding and, consequently, knowledge discovery.

A large number of different molecular representations have been devised, each showing a specific characteristics of a molecule (see Kozlíková et al. [KKF*17] for a recent survey of molecular visualization). For example, the ball-and-stick model shows the fundamental 3D structure and atomic bonds. Another example is the Cartoon representation for proteins, which provides an abstraction that shows the high-level, functional structure of a protein. For molecular interactions, surface representations that show the accessibility of a molecule with respect to another molecule (usually a solvent or ligand) are more suitable. One of the most useful molecular surface models for displaying accessibility is the SES (see Figure 1). The SES is defined by a probe sphere that rolls over the Van-der-Waals (VdW) spheres of a molecule. The radius of this probe is chosen so that it approximates a specific solvent molecule. The surface of the probe traces the SES, that is, parts of the molecule that are not reachable by the probe are not part of the SES. While rolling over the VdW spheres, the probe can be in three different states: (1) It can be in contact with just one VdW sphere while rolling. The part of the VdW sphere that is in contact with the rolling probe is a convex spherical patch that will be part of the SES. (2) When the probe is in contact with two VdW spheres, it can rotate around the axis defined by the two sphere centers, thereby tracing out a torus. The part of the torus that is between the two VdW spheres is a toroidal patch that is also part of the SES. (3) If the probe is in contact with three VdW spheres, it is in a fixed position, as it cannot roll any further without losing contact with one of the spheres. The concave spherical triangle on the probe surface between the three points of contact with the spheres is the last part of the SES. The leftmost image in Figure 1 shows the SES colored by patch type. The survey of Kozlíková et al. [KKF*17] gives a more detailed explanation of the SES.

The SES was defined by Richards [Ric77] using this rolling probe algorithm. A year later, Greer and Bush [GB78] gave another, equivalent definition: the SES can be defined by the union of all probe spheres that do not intersect the VdW spheres of the atoms. They also coined the term *Solvent Excluded Surface*, since the probes represent a certain solvent, which is excluded from the surface. Another name for the SES is Connolly surface, since Connolly was the first to present the analytical equations to compute the three surface patches [Con83]. Due to the complex nature of the SES, a naive computation is very time-consuming. Therefore, many algorithms accelerating this computations have been developed, which we briefly discuss in Section 2.

Richards' rolling probe algorithm also traces out a second type of molecular surface: the Solvent Accessible Surface (SAS) [Ric77]. The SAS is, however, defined by the center of the rolling probe. It is analogous to the VdW spheres, but the probe radius is added to the VdW radius. The SAS is computationally cheap, but it does not create a smooth surface and is, therefore, less suited for the analysis of molecular interactions [KKF*17].

In this paper, we present a fast algorithm to compute the SES in parallel on the GPU using Nvidia's Compute Unified Device Architecture (CUDA) for the computations and OpenGL/GLSL for rendering. Our approach employs the tremendous parallelization of the GPU to check for every triplet of atoms whether the probe sphere can be placed in a fixed position without intersecting any other atom. While this is a rather brute-force method, the computation time as well as the memory consumption on the GPU scale linearly with the number of atoms (in contrast to previous approaches, see Section 5.3). Our algorithm can render large molecular data sets interactively using a single consumer GPU. Our visualization of the SES is analytically correct and pixel-precise, since the surface is rendered using GPU-based ray casting [KBE09] (see Figure 1).

2. Related Work

The SES is the most commonly used surface representation, because it is suitable for the analysis of protein-solvent interaction or docking. The computation of the SES is, however, quite involved and time-consuming. Dynamic molecular data, such as simulation trajectories, typically contain several thousand to tens of thousands of frames. Thus, precomputing the SES for all frames—as done by some molecular viewers—is not a feasible approach for large, dynamic data. To enable the visualization of the SES for dynamic data at interactive frame rates, a method which computes the surface out-of-core is crucial. Hence, many approaches for the interactive computation and rendering of the SES have been proposed over the last 10 years. Below, we just briefly introduce the most notable and recent methods, an extensive overview of SES algorithms can be found in the survey by Kozlíkoá et al. [KKF*17], which covers the entire field of molecular structure visualization.

An efficient method to render the patches of the SES is the GPU-based ray casting presented by Krone et al. [KBE09]. The implicit description of the different patches is sent to the GPU, where shader programs are used to compute the ray-patch intersections. This method is well-suited, as it combines high image quality and low rendering times. Lindow at al. [LBPH10] presented a CPU-parallelized version of the Contour-Buildup algorithm by Totrov and Abagyan [TA95] to compute the SES interactively. Subsequently, Krone et al. [KGE11] adapted the Contour-Buildup to run efficiently on the GPU. Their implementation is to date one of the fastest ways to compute the SES analytically. Jurcik et al. [JPSK16] adapted their approach to enable semi-transparent SES rendering.

Another class of algorithms to compute the SES discretizes the space into voxel grids and samples the atoms to this grid. The SES can then be derived using an Euclidean distance transform [XZ09]. An efficient GPU implementation of this approach was recently presented by Hermosilla et al. [HKG*17]. By computing the grid progressively, they achieve interactive computation times even for very large molecular complexes. Egan and Gibou [EG18] presented a method that uses Octrees to refine the grid and that was implemented to run on massively parallel compute clusters. While the quality of the resulting SES is very good, the computation takes several seconds for larger data sets and is, thus, not interactive. In general, grid-based methods can usually be parallelized quite well and have low computation times. However, the quality of the SES is limited by the grid resolution. Therefore, we focused on the analytical computation of the SES.

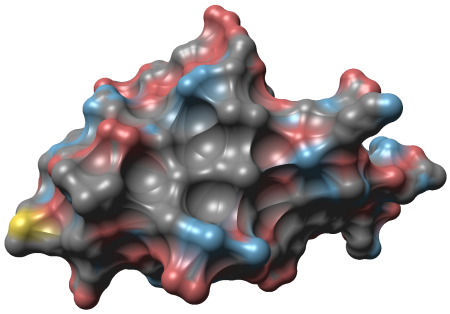

Figure 2: *Solvent Excluded Surface of an isomerase (PDB ID:* 1ogz*) colored by element. The probe radius was set to 2.4 Å.*

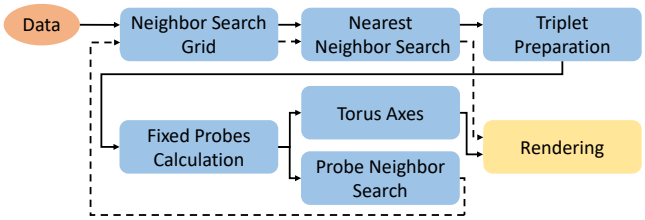

Figure 3: *Flow chart of our algorithmic pipeline. At the beginning, the data (orange oval) is uploaded to the GPU. All subsequent processing steps are executed entirely on the GPU by CUDA kernels (blue boxes). The final rendering step uses OpenGL (yellow box).*

3. Algorithm Overview

We present a new, highly parallel, analytic approach for the efficient calculation and rendering of the SES. In general, our algorithm computes all intersection points of three-way combinations of SAS spheres that represent the atoms of a protein. As mentioned in Section 1, the radius of a SAS sphere is the VdW radius of the corresponding atom plus the probe radius. These intersections represent the center of a probe sphere in a fixed position—that is, a probe in contact with three VdW spheres—which defines a spherical triangle. Only a fixed probe sphere that does not intersect with any other atom is in a valid position, all other fixed probe spheres are discarded. Our algorithm is specifically designed to exploit the massive parallel computational potential of modern GPU. The main part is the parallel computation of the intersection points (i.e., the valid fixed probes) of all combinations of three SAS spheres to derive the SES. From these intersections, the three graphical primitives that constitute the SES can be derived (spherical patches, spherical triangles, and toroidal patches). After this, the SES can be rendered. Figure 2 shows the SES of a protein.

To make use of the huge potential of modern GPUs, the algorithm has to be divided into multiple tasks with a moderate computational load. A GPU can perform a massive number of tasks in parallel in comparison to a CPU due to the much higher number of cores, but the resources per core are much more limited. The crucial Single Instruction Multiple Data (SIMD) architecture of GPUs is used to run an instruction in form of a CUDA kernel on multiple data via different threads [Fly72]. Therefore, all steps of our algorithm are designed to run in a highly parallel environment.

Figure 3 gives an overview of our algorithm. The first step of our algorithmic pipeline is to find all neighboring atoms of each atom. To do this efficiently, we need to construct a spatial acceleration structure. A grid-based data structure is a good choice since it maps well to the GPU and it can be computed quite fast, which is important when dealing with dynamic data. Using this neighbor search grid, we then execute a nearest neighbor search for each atom a_i. N_i is the set of all neighbor atoms a_j that are within a radius of $r_i + r_j + 2 \cdot r_p$, where r_i is the VdW radius of atom a_i, r_j is the VdW radius of atom a_j, and r_p is the probe radius. That is, N_i contains all neighboring atoms a_j that are no more than one probe diameter away from atom a_i, which means that the SAS spheres of the two atoms are at least touching each other.

The neighbor information is required for the next step, where we need to find all combinations of three SAS spheres that might intersect each other. For each atom a_i, two neighbors $a_j \in N_i$ and $a_k \in N_i$ with $i \neq j$ are chosen from the list of neighbors. If the SAS spheres of these atoms are also at least touching ($\|a_j - a_k\| \leq r_j + r_k + 2 \cdot r_p$), we have found a potential triplet of intersecting atoms. To calculate the two intersection points the spheres coordinates have to be transformed for S_i to $(0,0,0)$, for S_j to $(d,0,0)$ and for S_k to $(e,f,0)$ as a necessary simplification that allows to formulated the following three equations for the spheres [Mah15]:

$$r_1^2 = x^2 + y^2 + z^2 \tag{1}$$

$$r_2^2 = (x-d) + y^2 + z^2 \tag{2}$$

$$r_3^2 = (x-e) + (y-f) + z^2 \tag{3}$$

After the equations are solved, the coordinates have to be transformed back to get the correct coordinates [Mah15]. Both intersection points are potentially the centers of probes in a fixed position, as mentioned before. An alternative approach to compute the two potential fixed probe positions was given by Connolly [Con83]. To check if a probe position p is valid, it has to be tested for intersection with all other neighboring atoms $a_l \in N_i : l \neq j, l \neq k$. This intersection test is a simple distance test: $\|p - a_l\| > r_p + r_l$. Only valid fixed probe positions are stored for further processing, all others are discarded.

As mentioned above, the valid fixed probes found in the previous step define the spherical triangles. Each spherical triangle is mathematically defined by a fixed probe and the three corresponding atoms a_i, a_j, and a_k. To render the spherical triangles correctly, we also have to take care of the so-called *singularities*, which occur due to probe-probe intersections. These intersections can lead to spherical triangle patches that intersect each other. The singularities are the parts of the spherical triangles that are within the other probe and have to be removed. Therefore, we use a second nearest neighbor search to find all intersecting fixed probes for each fixed probe. Here, the nearest neighbor search radius is just $2 \cdot r_p$.

In addition, the information about the spherical triangles can be used to derive the toroidal patches: each pair of atoms (a_i, a_j), (a_i, a_k), (a_j, a_k) defines a torus axis. The analytic equations to compute the parameters for the tori were also given by Connolly [Con83]. Finally, the convex spherical patches that are part of the SES are simply the VdW spheres of the atoms. Thus, the

SES can be constructed by rendering the spherical triangles, the toroidal patches, and the VdW spheres of the atoms.

All steps of our algorithmic pipeline are designed to run in parallel on the GPU. In the next section, we describe a possible implementation of this algorithm.

4. Implementation Details

Our prototypical implementation of the algorithm described in Section 3 runs entirely on the GPU. We used CUDA for the computational part and OpenGL with GLSL shaders for rendering. Our implementation is designed for maximum speed, but also avoids data duplication and unnecessary copying of data. Especially memory transfer between host (CPU) and device (GPU) can still be a bottleneck. Therefore, the only data that is transferred to the GPU are the atomic positions and radii, and the per-atom colors. All further described operations use this data and derived data, which are only processed by the GPU.

4.1. Neighbor Search Grid

The first step after uploading the atomic properties (position, radius, color) to the GPU is to sort them into a grid for fast nearest neighbor retrieval. This neighbor search grid spans the entire bounding box of the molecule, that is, the maximum and minimum values of the Cartesian coordinates in (x,y,z) direction have to be determined. We use a uniform grid with cubic grid cells that have a side length

$$g_{dim} = 2 \cdot r_p + 4 \tag{4}$$

where r_p is the probe radius. The goal is to construct the grid such that a neighboring atom a_j satisfying the above mentioned neighbor criterion $\|a_i - a_j\| \leq r_i + r_j + 2 \cdot r_p$ is either located in the same grid cell as atom a_i or in one of the neighboring cells. Therefore, the factor of 4 is a conservative estimate of the sum of atom radii $r_i + r_j$ (since we are focusing on protein data sets, the VdW radius of all atoms will be below 2 Å).

Our neighbor search grid construction follows the *counting sort* CUDA implementation proposed by Hoetzlein [Hoe14]. A CUDA kernel that is executed in parallel for all atoms is used to assign the corresponding grid cell index to each atom (each grid cell is addressed via an individual hash value). In this step, the number of atoms per cell is determined by using an atomic operation for addition. This is necessary to guarantee the avoidance of race conditions, meaning that no other thread can increase the counter for the atoms before the current thread finishes its incrementation[†].

From the previous step, we have all the atoms and the assigned grid cell with the total count of atoms per cell. Next, the atoms have to be sorted into the assigned grid cells. Therefore, a parallelized prefix sum over the atom counts per cell is computed using the `inclusive_scan` function provided by the Thrust library[‡], which is part of the CUDA Toolkit. The following counting sort step uses this sum to put the atoms in the correct regions of the sorted array. The result of the prefix sum provides us with the array index range that is available for the atoms located in a specific grid cell. An atom a_i is assigned to cell g_j, which has the prefix sum s_j. That is, the atom a_i will be stored at position s_j and s_j will be decreased by one. Note that we only store the index of an atom in the sorted array to save memory. To later access the atoms located within a specific grid cell, we also need to store the start and end of each grid cell in a separate array. For more details about the counting sort grid construction, we refer to the original description by Hoetzlein [Hoe14].

4.2. Nearest Neighbor Search

We execute a nearest neighbor search in parallel for all atoms to get all neighboring atoms and write them to an array. Since the number of neighbors can be different for each atom, this step in theory requires a dynamic data structure. Since this would not map well to the CUDA API, we reserve space for 150 possible neighbors for each atom. This is an empirical number, which we found to be sufficient on our tests. For each atom a_i, the neighbors a_j are searched for in the grid cell that contains a_i and the directly neighboring cells ($3 \times 3 \times 3$ grid cells). As explained above, the neighbor criterion is defined as

$$\|a_i - a_j\| \leq r_i + r_j + 2 \cdot r_p \tag{5}$$

That is, each potential neighbor a_j found within one of the visited grid cells that satisfies this criterion is added to the list of actual neighbors. Two separate lists of neighbors are written: the first one is the complete list of all neighbors, the second one only contains neighbors where the atom index j is greater than the index i of the actual atom. This is an optimization to avoid redundant computations in later steps. The neighbors that satisfy the condition $j > i$ are referred to as *reduced neighbors* throughout the rest of the paper.

4.3. Preparation of SAS Sphere Triplets

The total number of reduced neighbors allows us to compute the number of theoretically possible combinations of three intersecting SAS spheres. In general, the number of possible k-tupels from a set of n items can be calculated using the combination formula $\binom{n}{k}$. In our case, we need to compute the number of triplets for a_i, that is, we need all possible combinations of two neighbors ($k = 2$). The number of theoretically possible SAS sphere triplets for a_i can then be calculated as follows:

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2} \tag{6}$$

where n is the number of neighbors of a_i. We compute this number of triplets in parallel for each atom and store the numbers in an array. Subsequently, a prefix sum over this array is computed using `thrust::inclusive_scan`. This prefix sum is again required to get the total amount of memory that we need to allocate to store the actual triplets as well as to get the array indices, where we store these triplets. The triplets are determined using the reduced neighbors lists and two nested for-loops. The first position of the combination is the current atom a_i and the remaining two positions

[†] CUDA Programming Guide: B.12. Atomic Functions https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions (last accessed 03/07/2019).

[‡] Thrust Parallel Algorithms Library https://thrust.github.io/ (last accessed 03/07/2019).

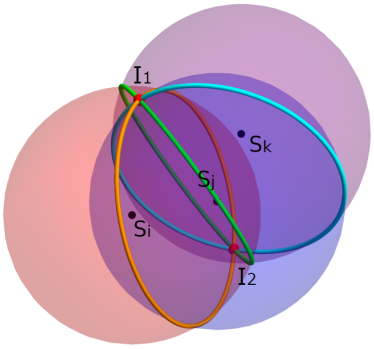

Figure 4: *Transparent spheres representing the SAS spheres of three atoms with their centers (S_i, S_j, S_k). The three colored circles show the course of the cut surfaces from the spheres. They meet each other in two points (I_1, I_2), which are the intersection points. (Image based on [Mah15])*

are filled with its neighbors. Note that we are avoiding redundant triplets by using the reduced neighbors. For example, if atom a_3 has the neighbors a_1 and a_6, we do not generate the triplet (a_3, a_1, a_6), since $1 < 6$ (that is, neighbor a_1 would not be available in the reduced neighbor list). This triplet was already generated for a_1 as triplet (a_1, a_3, a_6), which is sufficient since the order of the atoms does not matter for the following calculation.

4.4. Computation of Fixed Probe Positions

In this step, the actual intersection tests for the previously stored triplets of atom indices are computed. The define S_i, S_j, S_k as the SAS spheres corresponding to the atoms a_i, a_j, a_k, that is, they are at the same position but the radii of these spheres is the VdW radius of the atom plus the probe radius r_p. An obvious requirement for an intersection of three SAS spheres is that they all overlap with each other. Because we know that S_j and S_k are neighbors of S_i, we only have to test whether S_j and S_k overlap. Triplets that do not fulfill that criterion are excluded from further computations. Next, we check whether the two intersection points I_1 and I_2 (see Figure 4) for the current triplet (S_i, S_j, S_k) actually exist and compute their position as described in Section 3. For each intersection point, we also have to check whether it lies within another neighboring SAS sphere S_l. Here, the list of all neighbors has to be used, not just the reduced neighbor list (see Section 4.2). Only intersections that are outside of all neighbor SAS spheres are valid fixed probe positions that will be stored in a list for further processing.

In theory, we have to reserve enough memory to store both intersections for all triplets, which would require a large amount of memory. However, most of the triplets do not intersect at all or the intersection points are not valid probe positions, since they are cut away by neighboring SAS spheres. Therefore, the required size for the fixed probe position array is based on an empirically determined value in our implementation. In our tests, the final number of valid probe positions is less than one percent of the theoretical number of triplets (see Section 4.3). Consequently, we only allocate enough memory to store the fixed probe positions for one percent of the triplets. Since we compute the intersections of all triplets in parallel, we have to make sure that no intersection is lost due to race conditions between concurrent CUDA kernels. Therefore, we again use an atomic counter to store the fixed probe positions in the global array (see Section 4.1). If a valid fixed probe positions was found, the atomic counter is incremented and the position is written to the previously stored index.

4.5. Probe Neighbor Search for Singularity Handling

To avoid singularities of the spherical triangles in the final SES rendering, it is important to find fixed probes that are intersecting with neighboring fixed probes. Here, the same neighbor search procedure as for the atoms is used: the fixed probe positions are sorted into a uniform neighbor search grid (see Section 4.1), and then the actual neighbor search is performed (see Section 4.2). That is, for each probe p_i, all intersecting neighboring probes are stored. This list of probe neighbors will be used during rendering for the singularity handling of the spherical triangles.

4.6. Torus Axes

The torus axes are determined using the fixed probe positions. For each fixed probe, the atom index triplet (i, j, k) is stored. All three pairs of atoms define the axis of a torus (i.e., (a_i, a_j), (a_i, a_k), (a_j, a_k)). Thus, we write all these torus axes to an array using a CUDA kernel. After that, there are duplicates, since each combination will occur twice. To get rid of these duplicates, the torus axes are first sorted using `thrust::sort` and then the duplicates are removed by using `thrust::unique`, which removes consecutive duplicates in an array. Since all Thrust functions are running parallelized on the GPU, these steps are very fast.

4.7. GPU Memory Management and Rendering

We implemented our algorithm as a new plugin for the open source visualization framework MegaMol [GKM*15]. MegaMol already provides IO routines for molecular data (e.g., PDB file loading) and SES rendering using the fast GPU-based ray casting presented by Krone et al. [KBE09, KSES12]. Instead of triangulating the patches of the SES, special GLSL shaders compute the actual ray-object intersections during rendering, resulting in a pixel-perfect image of the patches. For the three types of SES patches (spherical, spherical triangles, toroidal), three different GLSL shaders are used. For details, please refer to the original publication [KBE09]. Below, we explain how we modified these shaders for our implementation.

Instead of using traditional Vertex Arrays (VA) or Vertex Buffer Objects (VBO), we store the data that has to be passed to the GLSL shader in Shader Storage Buffer Objects (SSBO). SSBOs have the convenient property that a GLSL shader has random access to the values stored in them. After allocating an SSBO via OpenGL, it can be registered by CUDA to get a resource (`cudaGraphicsGLRegisterBuffer`). This resource can then be mapped to CUDA (`cudaGraphicsMapResources`). From the mapped resource, a device pointer that allows read and write access to the SSBO memory from a CUDA kernel can be

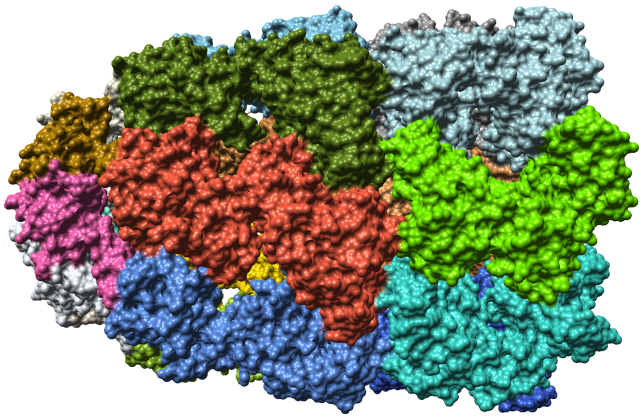

Figure 5: *Solvent Excluded Surface of a chaperonin complex (PDB ID:* 1aon*) for a probe radius of 1.4 Å.*

Table 1: *Performance measurements for different data sets.* Atoms: *number of atoms;* SES Share: *percentage of atoms that are actually part of the SES; overall* Run Time *of the SES calculation. Additionally, the required GPU memory (*VRAM*) and the frames per second (*FPS*) for rendering the final SES are listed. Note that the FPS include the time for recomputing the SES in each frame.*

Data Set	Atoms [10^3]	SES Share	Run Time [ms]	VRAM [GB]	FPS
1ogz	0.9	69%	6	0.12	118.7
1vis	2.5	60%	11	0.13	78.6
4x0l	4.2	64%	14	0.15	60.1
1af6	10.0	61%	26	0.18	33.1
1mmo	21.3	49%	82	0.28	12.0
1aon	58.7	68%	147	0.46	6.7
5tzs	98.5	92%	125	0.63	7.8
6hiv	99.4	71%	200	0.69	4.9
4v4j	147.1	69%	365	1.01	2.8
ribo01	147.2	65%	366	1.03	2.7
CCMV	214.4	62%	479	1.36	2.1

obtained (`cudaGraphicsResourceGetMappedPointer`). Prior to rendering, the resource must be unmapped again in order to be able to bind the SSBO for the GLSL shader. By writing all information that is required for rendering directly to a SSBO via CUDA kernels, we can completely avoid costly host-device memory copies or data duplication on the GPU. A further optimization is that the CUDA arrays and SSBOs are only resized if the current computation needs more memory, otherwise, the already allocated memory is re-used. To reduce the number of reallocations due to varying memory requirements (which can occur for dynamic data where the molecule deforms), all arrays and buffer objects are allocated with 10% excess. Since the amount of memory usually does not increase very much, this is usually sufficient to avoid a costly reallocation in every time step of a simulation.

The previous SES implementation available in MegaMol uses VA and VBO to transfer data to GPU memory for rendering. We modified all the existing shaders to use SSBOs. For the spherical patches, two SSBOs have to be created: one storing the atomic coordinates and VdW radii, and one for the atom colors. These data is used to render all atoms as colored spheres, that is, the spherical patches that are part of the SES.

Previously, all parameters required for rendering the spherical triangles and toroidal patches were precomputed and also passed to the GPU as VA or VBO. For our new SES rendering, we compute these parameters in the vertex and fragment shaders, which reduces the required amount of memory. As mentioned in Section 3, the necessary equations were given by Connolly [Con83]. The data for the spherical triangles consists of five SSBOs: atom positions + radii, atom colors, fixed probe positions, the corresponding triplets, and the list of probe neighbors for singularity handling. Note that the SSBO for the triplets contains only the atom indices through which the actual positions stored in the first SSBO can be accessed. The rendering of the toroidal patches requires three SSBOs: atom positions + radii, atom colors, and the torus axes. The torus axes SSBO also just stores atom indices. The two SSBOs containing the atom positions + radii and the colors can of course be re-used by all shaders, thus further reducing the amount of memory that has to be transferred to the GPU for rendering.

5. Results & Discussion

We measured the performance of our implementation on a test system running Windows 10 and equipped with an Intel i5-8600k CPU ($6 \times \sim$3.6 GHz), 16 GB RAM, and a NVIDIA Geforce GTX 1080 (8 GB VRAM, 2560 CUDA cores). Table 1 shows the timings for test data sets of various sizes obtained from the PDB [BWF*00]. The resolution was set to FullHD (1920×1080) to measure the rendering performance. Please note that we treated the data as if it was dynamic, that is, the whole SES computation was run in each frame. Only the memory allocation was not done for every iteration. Our method is able to compute the SES interactively for molecular complex of more than 20 k atoms (maintaining a frame rate of more than 12 fps on our test system). Even for our largest test data set (CCMV), a virus capsid of more than 200 k atoms, our implementation takes less than 500 ms for the SES computation, consequently reaching more than 2 fps. Figure 5 shows one of our test data sets, a chaperonin complex with ∼58 k atoms.

Table 2 and Figure 6 show a more detailed breakdown of the timings for the individual steps of our algorithmic pipeline. As observable, the parallel computation of fixed probe positions, which is the core of our algorithm, is the most time-consuming step. This is not surprising, since it has to check all the possible triplets of SAS spheres for each atom. The other parts play a subordinate role. On average, preparing the triplets and computing the fixed probes takes more than 80% of the whole computation time. As observable, the total runtime of our algorithm scales linearly with the number of atoms. An exception is the data set *5tzs*, which has a very special atomic configuration (see Figure 7) and, therefore, exhibits an irregular runtime. We will discuss this outlier case in Section 5.1.

We also measured the VRAM consumption (see Table 1). In general, the memory requirements of our method are reasonably low

Table 2: *Timings for the individual steps of our algorithmic pipeline (all measurements in milliseconds).* Grid*: inserting the atoms into the grid for neighbor search,* Neighbor*: nearest neighbor search,* Triplet*: preparing the list of potentially intersecting three SAS spheres,* Probe*: find valid fixed probe positions,* Tori*: derive torus information from fixed probe positions,* PN*: search all neighboring probes for each fixed probe position.*

Data Set	Grid	Neighbor	Triplet	Probe	Tori	PN
1ogz	2.04	0.89	0.70	1.38	0.32	0.94
1vis	3.58	1.00	0.92	3.98	0.39	1.05
4x0l	4.00	0.96	0.80	6.90	0.35	1.30
1af6	4.68	0.96	1.81	17.06	0.36	1.53
1mmo	5.91	1.68	12.30	59.70	0.43	1.88
1aon	19.29	3.86	28.44	88.63	1.17	5.72
5tzs	21.14	3.73	25.72	58.21	2.04	14.39
6hiv	13.28	5.72	43.40	126.64	1.70	9.61
4v4j	26.74	9.08	80.50	231.53	2.25	14.97
ribo01	16.43	9.48	79.69	244.34	2.32	13.36
CCMV	31.71	13.23	103.68	309.49	2.94	18.15

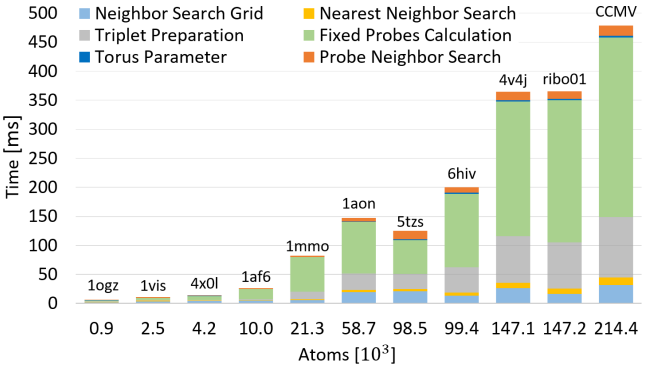

Figure 6: *The atoms per molecule [10^3] are plotted against the time for SES calculation [ms]. Each stacked bar is labeled with the PDB ID of the RSCB protein data bank.* ribo01 *is a ribosome consisting of 2WDL and 2WDK,* CCMV *is a virus capsid consisting of multiple 1CWP units.*

and also depend linearly on the number of atoms, except for data set *5tzs*. This outlier is again due to the fact that the memory consumption also depends on the conformation of the atoms within the molecule (see Section 5.1 for details).

5.1. Conformational Dependency

As mentioned above, the runtime and memory consumption increases linearly with the number of atoms for our test data sets (see Table 1), with the exception of data set *5tzs*. Here, the computation time drops by about 17% compared to *1aon*, although *5tzs* has ∼68% more atoms. The computation time of the only ∼1% larger data set *6hiv* is even about 60% higher than the one of *5tzs*. A closer look at Table 2 reveals that the main difference is due to the considerably shorter execution time of the step that prepares the triplets and finds the valid fixed probe positions. This can be explained by the conformation (i.e., the atomic configuration) of *5tzs*. This data set has more atoms as well as a higher relative number of atoms that contributes to the SES: 92% of the atoms in *5tzs* contribute to the SES (see *SES Share* in Table 1) compared to only 68% for *1aon*. Additionally, *5tzs* also has a larger relative spatial extent with respect to the total number of atoms. This larger spatial extent and the higher *SES Share* are due to the fact that this data set is much sparser than a typical protein complex (having a lot of empty space in between) and that it contains long strands of Ribonucleic Acid (RNA) that stick out (see Figure 7). These two aspects result in a configuration where the atoms have fewer neighbors.

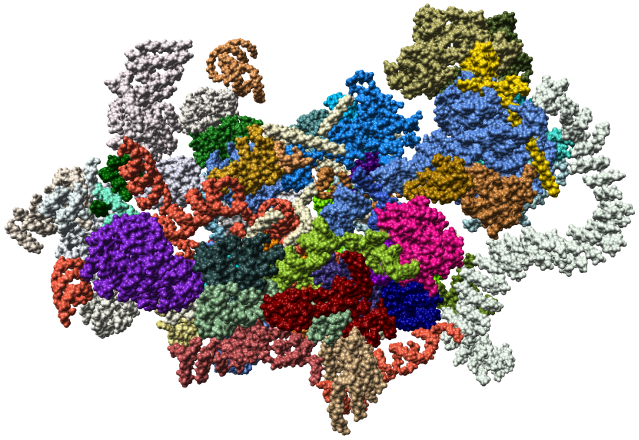

Figure 7: *SES of the yeast small subunit processome [CMBHK17] (PDB ID:* 5tzs*, $r_p = 1.4\,Å$). Each chain is colored differently.*

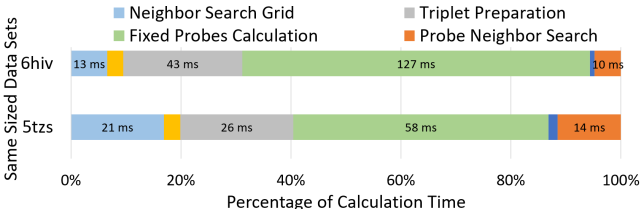

Figure 8: *Relative time consumption [%] of the different parts of our SES algorithm compared to the total time. The two similar-sized data sets* 6hiv *and* 5tzs *are shown. The bar length of the individual algorithm parts represents the relative time consumption. Additionally, the labels show the absolute time [ms] for the crucial parts. See Figure 6 for a complete description of color usage.*

The lower number of neighbors per atom for *5tzs* lowers the run time for preparing the triplets and finding the fixed probes compared to the similar-sized *6hiv* (see Figure 8). The number of triplets depends quadratically on the number of neighbors (see Equation 6). Consequently, a lower number of neighbors leads to much fewer triplets per atom. That is, much fewer triplets have to be checked to find the valid fixed probe positions. Furthermore, for each potential probe position, much fewer neighboring atoms have to be checked for intersection to make sure that the probe is in

a valid position. The slightly longer calculation time for the fixed probe neighbor search can be explained by the higher total number of valid fixed probes: since the atoms are distributed more sparsely in *5tzs*, more triplets will actually create valid fixed probe positions.

5.2. Discussion of the Optimization Process

In this section, we discuss the process of optimizing our implementation with respect to memory consumption as well as computation speed. We tried different approaches until we reached the final solution described in Section 4.

1. Exact Method: We tested different approaches for allocating the minimally required GPU memory. To do so, we had to run the calculation routine for finding fixed probe positions two times. In the first run, the triplets for each atom were examined regarding the fixed probes and the number of valid fixed probes was stored for each atom. Then a prefix sum was calculated to get the total number of fixed probes, which determines the amount of memory that needs to be allocated in order to actually store all valid fixed probes in a second run of the computation. Using this method, only the necessary memory size was reserved, but as the fixed probe calculation is the most time-consuming part of our algorithm, running it two times was too expensive.

2. Sort And Unique Method: An array twice the size of the number of all triplets was allocated, which is sufficient to store both possible intersections for each triplet—that is, potentially all possible fixed probe positions. Next, the routine for the fixed probe calculation was executed only once, directly writing only the valid fixed probe positions to the array. After that, this solution array was sorted to get all empty entries of the array to a coherent block prior to running `thrust::unique`. This compactifies the array and gives the number of fixed probes. This method was faster than the first approach, but requires a huge amount of memory.

3. Heuristic AtomicAdd Method: The third method is the one described in Section 4.4, which uses `atomicAdd` to write valid fixed probe positions. When comparing all three approaches, we found that using atomic operations is actually the fastest way to solve the problem. So, this approach not only exhibits the best performance, the memory consumption is also very similar to the first, due to the heuristic estimation of required memory.

5.3. Comparison with Previous Work

We compared our method to the GPU-parallelized Contour-Buildup algorithm by Krone et al. [KGE11], which is available in the open source visualization framework MegaMol [GKM*15]. The computation speed of the Contour-Buildup is only slightly faster than our method for small data sets. However, for larger data sets, the Contour Buildup is considerably faster and exhibits better scaling (see Table 3, *Performance*). However, the Contour Buildup implementation requires a disproportional amount of GPU memory with increasing numbers of atoms (see Table 3, *VRAM*). The chaperonin *1aon* was actually the largest of our test data sets that we were able to visualize using the Contour-Buildup, as the GPU ran out of memory for the larger ones. For this data set, our implementation still requires only ∼460 MB of GPU memory, while the Contour-Buildup requires more than 10× the memory (5 GB). That is, the memory requirements of the CUDA Contour-Buildup implementation currently available in MegaMol are comparable to our second approach described in Section 5.2, which makes it unfeasible for large data sets even though the computation speed would still be sufficient. In contrast, our implementation still requires only 1.36 GB of VRAM even for the largest of our test data sets, the virus capsid with more than 214 k atoms shown in Figure 1 (right).

Table 3: *Comparison of the overall performance and memory requirements (*VRAM*) of our implementation and the CUDA Contour-Buildup (*CB*) presented by Krone et al. [KGE11].*

Data Set	Atoms	Performance		VRAM	
	[10^3]	ours	CB	ours	CB
1vis	2.5	78.6 fps	86 fps	0.13 GB	0.4 GB
1af6	10.0	33.1 fps	51 fps	0.18 GB	1 GB
1aon	58.7	6.7 fps	20 fps	0.46 GB	5 GB

6. Summary & Outlook

We presented a new massively parallel algorithm for the computation of an analytically correct SES on the GPU. Our algorithm was mainly designed to exhaust the parallel computation capacity of modern graphics hardware in order to handle dynamic data, which requires the re-computation of the SES in each frame that is rendered. Our prototypical CUDA implementation maintains interactive frame rates for molecules of more than 20 k atoms. The algorithm is fast, but not the fastest available implementation of a SES calculation, however, the much lower VRAM consumption facilitates loading larger data sets and enables machines with lower VRAM to compute the SES. As discussed in Section 5.1, the calculation time depends on the spatial extents, and structural features like cavities and tunnels that create void spaces, and, consequently, on the number of atoms that actually contribute to the SES. However, for a typical protein, the computation time as well as the memory consumption scales linearly with the number of atoms.

In the future, our method could be combined with fast, approximate molecular surface calculations: Areas of interest such as cavities and tunnels would be computed with the analytically exact SES method and the remaining molecular surface could be calculated using a fast, approximating method like Gaussian molecular surfaces [KSES12] to speed up the whole procedure, similar to the approach of Parulek et al. [PJR*14]. A further possible extension would be to use probes with different radii, depending on the possible solvents or ligands that can interact in certain areas of the molecule. That is, the SES at interaction area x for ligand l_x would be computed with a probe radius r_x, whereas area y for ligand l_y would be computed with a probe radius r_y. The remaining surface parts could be calculated using a generic probe radius (e.g., water).

Acknowledgments

This work was partially funded by German Research Foundation (DFG) within project PROLINT.

References

[BWF*00] BERMAN H. M., WESTBROOK J., FENG Z., GILLILAND G., BHAT T. N., WEISSIG H., SHINDYALOV I. N., BOURNE P. E.: The Protein Data Bank. *Nucleic Acids Research 28*, 1 (2000), 235–242. URL: http://www.pdb.org, doi:10.1093/nar/28.1.235. 1, 6

[CMBHK17] CHAKER-MARGOT M., BARANDUN J., HUNZIKER M., KLINGE S.: Architecture of the yeast small subunit processome. *Science (New York, N.Y.) 355*, 6321 (2017). doi:10.1126/science.aal1880. 7

[Con83] CONNOLLY M. L.: Analytical Molecular Surface Calculation. *Journal of Applied Crystallography 16*, 5 (1983), 548–558. doi:10.1107/S0021889883010985. 2, 3, 6

[EG18] EGAN R., GIBOU F.: Fast and scalable algorithms for constructing Solvent-Excluded Surfaces of large biomolecules. *Journal of Computational Physics 374* (Dec. 2018), 91–120. doi:10.1016/j.jcp.2018.07.035. 2

[Fly72] FLYNN M. J.: Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (1972), 948–960. doi:10.1109/TC.1972.5009071. 3

[GB78] GREER J., BUSH B. L.: Macromolecular shape and surface maps by solvent exclusion. *Proceedings of the National Academy of Sciences 75* (1978), 303–307. 2

[GKM*15] GROTTEL S., KRONE M., MÜLLER C., REINA G., ERTL T.: MegaMol - A Prototyping Framework for Particle-based Visualization. *IEEE Transactions on Visualization and Computer Graphics 21*, 2 (2015), 201–214. doi:10.1109/TVCG.2014.2350479. 5, 8

[HKG*17] HERMOSILLA P., KRONE M., GUALLAR V., VÁZQUEZ P.-P., VINACUA L., ROPINSKI T.: Interactive GPU-based generation of solvent-excluded surfaces. *The Visual Computer 33*, 6 (2017), 869–881. doi:10.1007/s00371-017-1397-2. 2

[Hoe14] HOETZLEIN R. C.: Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. Nvidia GPU Technology Conference (talk), 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf. 4

[JPSK16] JURCIK A., PARULEK J., SOCHOR J., KOZLIKOVA B.: Accelerated Visualization of Transparent Molecular Surfaces in Molecular Dynamics. In *IEEE Pacific Visualization Symposium* (2016), pp. 112–119. doi:10.1109/PACIFICVIS.2016.7465258. 2

[KBE09] KRONE M., BIDMON K., ERTL T.: Interactive Visualization of Molecular Surface Dynamics. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1391–1398. doi:10.1109/TVCG.2009.157. 2, 5

[KGE11] KRONE M., GROTTEL S., ERTL T.: Parallel Contour-Buildup Algorithm for the Molecular Surface. In *IEEE Symposium on Biological Data Visualization* (2011), pp. 17–22. doi:10.1109/BioVis.2011.6094043. 2, 8

[KKF*17] KOZLÍKOVÁ B., KRONE M., FALK M., LINDOW N., BAADEN M., BAUM D., VIOLA I., PARULEK J., HEGE H.-C.: Visualization of Biomolecular Structures: State of the Art Revisited. *Computer Graphics Forum 36*, 8 (2017), 178–204. doi:10.1111/cgf.13072. 2

[KSES12] KRONE M., STONE J. E., ERTL T., SCHULTEN K.: Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis - Short Papers* (2012), pp. 67–71. doi:10.2312/PE/EuroVisShort/EuroVisShort2012/067-071. 5, 8

[LBPH10] LINDOW N., BAUM D., PROHASKA S., HEGE H.-C.: Accelerated Visualization of Dynamic Molecular Surfaces. *Computer Graphics Forum 29*, 3 (2010), 943–952. doi:10.1111/j.1467-8659.2009.01693.x. 2

[Mah15] MAHIEU E.: Trilateration and the intersection of three spheres: http://demonstrations.wolfram.com/TrilaterationAndTheIntersectionOfThreeSpheres/: Wolfram demonstrations project. 3, 5

[PJR*14] PARULEK J., JÖNSSON D., ROPINSKI T., BRUCKNER S., YNNERMAN A., VIOLA I.: Continuous Levels-of-Detail and Visual Abstraction for Seamless Molecular Visualization. *Computer Graphics Forum 33*, 6 (2014), 276–287. 8

[Ric77] RICHARDS F. M.: Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering 6*, 1 (1977), 151–176. doi:10.1146/annurev.bb.06.060177.001055. 2

[TA95] TOTROV M., ABAGYAN R.: The Contour-Buildup Algorithm to Calculate the Analytical Molecular Surface. *Journal of Structural Biology 116* (1995), 138–143. doi:10.1006/jsbi.1996.0022. 2

[XZ09] XU D., ZHANG Y.: Generating Triangulated Macromolecular Surfaces by Euclidean Distance Transform. *PLOS ONE 4*, 12 (Feb. 2009), e8140. doi:10.1371/journal.pone.0008140. 2