Computing and Informatics, Vol. 29, 2010, 1325–1336

FASTGRID – THE ACCELERATED AUTOGRID POTENTIAL MAPS GENERATION FOR MOLECULAR DOCKING

Marek Olšák, Jiří Filipovič

Faculty of Informatics, Masaryk University
Botanická 68a
602 00 Brno, Czech Republic
e-mail: {mareko, fila}@mail.muni.cz

Martin Prokop

National Centre for Biomolecular Research
Faculty of Science, Masarych University
Kotlářská 2
611 37 Brno, Czech Republic
e-mail: martinp@chemi.muni.cz

Revised manuscript received 11 May 2010

Abstract. The AutoDock suite is widely used molecular docking software consisting of two main programs – AutoGrid for precomputation of potential grid maps and AutoDock for docking into potential grid maps. In this paper, the acceleration of potential maps generation based on AutoGrid and its implementation called FastGrid is presented. The most computationally expensive algorithms are accelerated using GPU, the rest of algorithms run on CPU with asymptotically lower time complexity that has been obtained using more sophisticated data structures than in the original AutoGrid code. Moreover, the CPU implementation is parallelized to fully exploit computational power of machines that are equipped with more CPU cores than GPUs. Our implementation outperforms original AutoGrid more than 400× for large, but quite common molecules and sufficiently large grids.

Keywords: Molecular docking, acceleration, CUDA, AutoDock

1 INTRODUCTION

Recent advances in the hardware architecture of graphics processing units (GPUs) made it applicable for high performance scientific computing. Many of GPU computing applications belong to the field of computational chemistry where they are used mainly for quantum chemical calculations [3] and molecular dynamics [1, 4]. Molecular docking is another field of computational chemistry with great potential for GPU computing utilization.

Molecular docking is a method for prediction of a position and an orientation of one molecule to another when bound to each other to form a stable complex. One of the categories of molecular docking, which is important in computer-aided drug design, is the protein-ligand docking where a small molecule of a ligand (putative drug) is docked into a binding site of a protein. Calculation of interaction energy between the protein and the ligand is the most computationally demanding part of docking algorithms. The interaction energy is usually calculated as a sum of pairwise interactions between the ligand and protein atoms. Two main types of atom-atom interactions are electrostatic and van der Waals (vdW) interactions, even though some docking programs also use interactions describing hydrogen bonds and solvation effects. Van der Waals interactions are short-range, they decay as the inverse sixth power of an atom-atom distance. Hence vdW interactions can be neglected for atoms that are too distant in space and are calculated only for atoms closer than a user-specified cutoff distance decreasing computational requirements. Electrostatic interactions, calculated from atomic partial charges using Coulomb's law, are long-range, they decay as the inverse of an atom-atom distance. The cutoff approach is thus not applicable here. For large systems with ten thousands of atoms, fast approximation methods were developed [5, 6, 7] but for mid-ranged systems of few thousands of atoms, which are typically used in protein-ligand docking studies, electrostatic interactions must be calculated using summation over all atoms (called the direct summation algorithm). Although this is computationally demanding, it is well suited for parallel algorithms.

One of the most widely used software for protein-ligand docking is the AutoDock program [2, 8]. It uses genetic algorithms to find a position, an orientation or a conformation of a ligand molecule in a binding site of a protein (or other large molecule such as DNA). The molecule of the protein is treated as a rigid body which enables to precalculate electrostatic and vdW potentials on a grid. The interaction energy of the ligand atom with protein atoms can be calculated from a linear interpolation of the values of grid points. AutoDock uses the AutoGrid program to calculate electrostatic and vdW potential grid maps. The algorithm implemented in AutoGrid is not well optimized and it does not utilize a GPU. Since the calculation of grid maps is a computationally demanding task, especially for big or dense grids, an implementation of optimized algorithms which would utilize the GPU power is desirable.

We present FastGrid software which implements fast algorithms for the calculation of electrostatic and vdW potential grid maps here. The electrostatic potential is calculated using a parallel algorithm utilizing GPU hardware. The calculation of other potentials was improved by implementing a fast algorithm for finding atoms within a specified distance range.

2 POTENTIAL MAPS GENERATION

The AutoGrid generates an electrostatic grid map and multiple non-electrostatic potential maps (including the desolvation, van der Waals, and the hydrogen-bond potential), one for each atom type. It also generates a map containing the charge-dependent term of desolvation potential. Each grid map is a three-dimensional lattice of regularly spaced points. For each point, the potential energy of the probe atom given by the molecule is computed.

Algorithm 1 Scheme of the AutoGrid main loop

```
 1: for all points of the grid do
 2:    for all atoms of the molecule do
 3:       compute the electrostatic potential
 4:       if the distance between the point and the atom < the cutoff value then
 5:          compute the desolvation potential
 6:          compute the van der Waals potential
 7:          compute the hydrogen-bonds potential
 8:       end if
 9:    end for
10: end for
```

Algorithm 1 sketches the main loop of AutoGrid, whose efficiency is crucial for overall AutoGrid performance. From the computational point of view, there are two kinds of energies:

- The long-range energies influence all points of the grid, thus for a $n \times n \times n$ grid and m atoms, the problem complexity is $\mathcal{O}(m \cdot n^3)$.
- The short-range energies use the cutoff approximation (i.e. only near atoms are used for energy evaluation) yielding $\mathcal{O}(n^3)$ time complexity.

The electrostatic potential evaluation is most critical for large molecules because of its time complexity. However, its computation is very regular (i.e. suitable for SIMD processors), thus it can be computed using a GPU very efficiently [1]. The rest of potentials is computed only for the near atoms (i.e. closer than the cutoff distance) and we leave these computations on a CPU, which has two reasons: first, these computations yield an irregular memory access pattern, which is not very suitable for a GPU; second, the CPU and the GPU should work concurrently to fully exploit the computational power of a system.

One of the demanding tasks in the main program loop is the calculation of the distance between a grid point and an atom. This value is used to evaluate the electrostatic potential and it is also necessary in other potentials calculations for finding atoms closer than the cutoff distance. In the original AutoGrid code, the value is calculated within the *compute the electrostatic potential* part of the loop and is further available for other parts of the loop where the cutoff atoms are searched. Moving the electrostatic potential calculation from the CPU to the GPU makes the distance value unavailable for other parts of the loop calculated on the CPU. Calculating the distance once again on the CPU would increase the time complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(m \cdot n^3)$. Although the cutoff atoms can be searched during the GPU computations, it has two disadvantages; first, the GPU algorithm has to build a list of found atoms which influences its performance; second, the CPU and GPU computations cannot overlap easily. The optimal solution to the problem is an implementation of advanced algorithm for finding atoms closer than the cutoff distance.

3 MAP COMPUTATIONS USING CUTOFF

3.1 The Cutoff Grid

The key requirement for the data structure is the ability to obtain a list of atoms located in the cutoff distance from a given lattice point, ideally in time complexity $\mathcal{O}(1)$. Therefore, we introduce a 3-dimensional grid having the same spatial size and position in space as the potential map but is considerably coarser. The grid is divided into a set of regularly-sized rectangular boxes. Each box represents a bucket containing atoms that lie in the cutoff distance from it. Thus, an atom may fall into several neighboring buckets, making them slightly overlap each other. The memory address of the bucket is directly computed from a position in space.

Creation of the grid is based on a sphere-box intersection where each atom is represented as a sphere with a radius being the cutoff distance, and each bucket is represented as a box. If the sphere and the box intersect, the atom falls in the bucket. To reduce the number of possible intersections, each sphere is bounded by another box whose opposite corners denote a 3D range of buckets and only those are tested with the particular sphere. This implies that the maximum amount of intersections which need to be computed for one atom can be directly derived from the cutoff distance, which is a constant. Therefore, the time complexity for creating the grid is $\mathcal{O}(n)$ where n is the number of atoms.

The original algorithm for calculating potential energies in a given lattice point, where fine-grained cutting off takes place, is then performed on the obtained list of atoms from the corresponding bucket instead of being performed on the whole molecule. Therefore, the overall asymptotic time complexity of the potential evaluation depends on the grid size and atoms density in the near neighborhood of the lattice point (which is upper bounded by a constant).

3.2 Hydrogen Bonds Potential

The part of the hydrogen bonds potential computation is searching for the closest hydrogen from a lattice point, which was originally implemented using a linear search. We have rewritten it using kd-tree data structure [9] filled with hydrogens only. This solution yields time complexity of searching $\mathcal{O}(\log m \cdot n^3)$ for m hydrogens and a $n \times n \times n$ grid.

Using the cutoff grid would not help here: If a bucket at a given point does not contain a hydrogen, looking in the neighboring buckets and comparing the closest hydrogens from each of them in linear time would not be very efficient.

3.3 Parallelism

In order to fully exploit the power of multiple CPU cores, which are commonly available in the today's machines, we decided to use OpenMP[1] – the C compiler extension that facilitates the use of multi-threading.

In FastGrid, a potential map is divided into slices perpendicular to the Z direction, having the same width and height as the potential map and being 1 lattice point thick. Each thread then takes a not yet processed slice and computes it. This approach generates small enough jobs to maintain proper load balancing across all CPU cores for the algorithm to scale well, while the overhead of assigning jobs to threads is almost unmeasurable.

4 ELECTROSTATIC MAP COMPUTATION ON THE GPU

In this chapter, brief overview of CUDA architecture is given and the GPU implementation of the electrostatic potential computation is described.

4.1 GPU Architecture and CUDA

For our purposes, we have chosen CUDA-enabled GPUs because of their performance and sufficient programmability. Detailed description of nVIDIA GPU architecture and CUDA is out of the scope of this paper and can be found in [10], we provide here only the brief overview of the CUDA threading and memory model necessary to understand the implementation of electrostatic potential calculation.

The today's hi-end GPUs consist of tens of multiprocessors having access into global memory. Each multiprocessor is equipped with multiple ALU cores performing the same instruction at a time. Thus, a set of threads (called a warp) should follow the same program path to fully utilize the multiprocessor. The code within a warp can diverge, however, the different branches are performed in serial. Because the GPU executes instructions in order, hiding of memory latencies is given by switching warps (which is implemented in hardware) – if a warp waits for a memory

[1] `http://www.openmp.org`

transaction, the GPU can run another warp. Because of the memory latency-hiding mechanism, the number of threads running on the GPU should be an order of magnitude higher than the number of GPU cores.

In CUDA, code for the GPU is written as a kernel function, which is executed as a GPU thread multiple times in parallel. The GPU threads are organized into a thread block and the thread blocks are organized into a block grid. All threads within one block can communicate via shared memory and can be synchronized via hardware implemented barrier. The grid consists of multiple thread blocks running the same code. This model provides a natural and scalable mapping to the GPU hardware architecture. All threads in one thread block are executed on the same multiprocessor, allowing them to use its shared resources. For a good utilization of the multiprocessor resources, the block size has to be a multiple of a warp size and a few warps have to run on each multiprocessor. For a good utilization of the GPU, the sufficient number of blocks have to be executed to utilize all multiprocessors.

The GPU memory subsystem is very fast, but most memory access models are uncached and the code has to meet some restrictions to exploit the huge memory bandwidth or short latencies of some types of memory. The global memory is accessible from all multiprocessors for reading and writing. It is an order of magnitude faster than system RAM if it is accessed coalesced. The latency of operations in global memory is in hundreds of GPU cycles. The texture memory is read-only memory whose caching is optimized for a 2D spatial locality. The latency of the cache hit in texture memory is high, thus it can be used to improve the bandwidth when data is accessed multiple times or the data access is not regular enough to exploit the bandwidth of global memory. The constant memory is read-only and cached too, but the latency of the cache hit is as short as accessing a register if all threads within one warp access the same address. Thus, it can be used for constants that are accessed by many threads in parallel.

Currently, the arithmetic throughput and the memory bandwidth of GPUs are an order of magnitude higher than CPU ones. Moreover, the GPUs have specialized cores for some operations used frequently in graphics calculations and thus can improve performance even more when these operations are used.

4.2 Implementation

As a basis, we have used the published acceleration of the direct Coulomb summation [1]. The potential map resides in global memory, because its access pattern can be easily coalesced. Considering only one of the atoms is accessed at a time, we place them in constant memory and because of its limited size, just over 4 000 atoms fit there, so several kernel invocations are required. The GPU thread calculates 1–8 lattice points (see below), sums up potential energies of atoms, and stores them in the lattice.

The method discussed in [1] computes only one slice of the lattice per a kernel invocation, but this approach does not scale well on small maps. Therefore, we execute the kernel on the whole lattice to allocate enough thread blocks on multiprocessors. Because the GPU can run a thread block as a whole, the initialized potential map must be padded to the size of a portion of map a thread block computes. It is more efficient to compute lattice points lying outside of the original potential map than explicitly handling boundary conditions, considering the GPU cannot handle divergent code branching efficiently in general.

The loop over lattice points can be unrolled by 8 in the Z direction, allowing the computation of 8 lattice points per thread, which increases performance as some of the computations can be shared. Unrolling the outer loop in the Z direction is the only viable option, otherwise doing it in other directions would waste too much space and computational resources due to padding to the thread block size and the small maps would not perform as well as the large ones. We disable unrolling if the cost of padding in the Z direction exceeds 150% of the original potential map. The thread block allocates 16×8 threads, and based on the fact whether unrolling is enabled or not, it computes the lattice segment of size either $16 \times 8 \times 8$ or $16 \times 8 \times 1$.

4.3 Dielectrics Computation

AutoGrid uses constant and distance-dependent dielectrics. The implementation of the former is straightforward and is described in [1].

We have implemented two variants of the distance-dependent dieletric computation on the GPU:

- The first variant uses a precalculated look-up table placed in texture memory and indexed by the distance from an atom. Because the distances from a lattice point are similar for consecutive atoms, it gives a good spatial locality in texture accesses if the distances between lattice points are not too large. In this case, it provides the best performance in comparison with other memory types for sufficiently large gridmaps.
- The second variant directly computes dielectrics in the code. It is used in cases where the texture memory approach is slower, especially with small gridmaps.

The FastGrid chooses between the two automatically. Because the efficiency of texture memory depends on the longest edge of the lattice rather than the overall number of lattice points, FastGrid uses the second variant if the inequality $\max(x, y, z) < 160$ holds (where x, y, z are the lattice dimensions), which works for most sizes of potential maps.

5 EVALUATION

The FastGrid has been evaluated using a machine equipped with Intel Core2 Quad Q9550 (2.83 GHz), 8 GB RAM and two GeForce GTX 280 connected via PCI-E 2.0 8x. The half of the Collagen, type IV, alpha 2 structure (PDB[2] ID 1t60, 25 448 atoms) has been used for benchmarking.

[2] `http://www.pdb.org/pdb/`

5.1 The Speedup Over AutoGrid

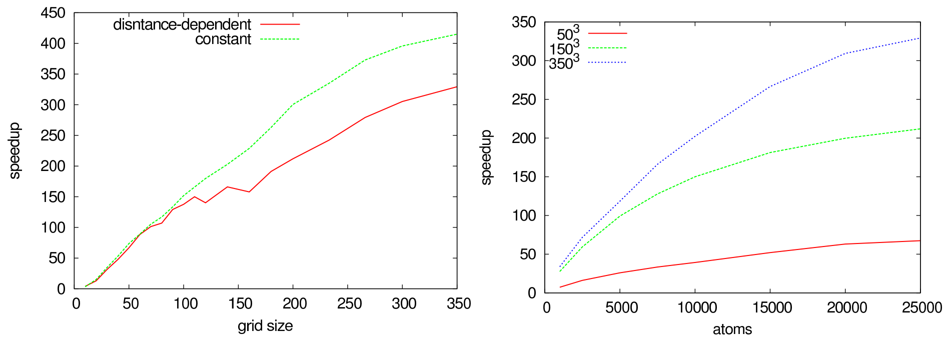

Fig. 1. The speedup of FastGrid over AutoGrid. Left: using various grid sizes, right: using various numbers of atoms for $50 \times 50 \times 50$, $150 \times 150 \times 150$ and $350 \times 350 \times 350$ grids and distance-dependent dielectrics.

Figure 1 depicts the speedup of our implementation over original AutoGrid and its scaling according to the grid size and the number of atoms. The FastGrid outperforms AutoGrid in an order of magnitude even for quite small grids or quite small molecules if a finer grid is used.

However, because AutoGrid uses only one CPU thread, its performance can be better if multiple problems are solved in parallel on a multi-core CPU. If multiple GPUs are present in the system, it is meaningful to run multiple instances of Fast-Grid too, if the GPU is the bottleneck, which is analyzed in the following chapter.

5.2 Analysis of FastGrid Performance

A deeper analysis of FastGrid performance helps us find a well-balanced machine configuration and can help the decision whether multiple instances of FastGrid using multiple GPUs should run at the same time to improve the overall performance. Because of higher time complexity of the GPU computations, the GPU performance is the bottleneck for larger molecules. According to the results depicted in the left graph in Figure 2, the multiple instances of FastGrid using multiple GPUs are meaningful only when distance-dependent dielectrics is used with the given molecule and the machine. If less powerful GPUs or larger molecules are used, multiple instances can be more beneficial. The GPU performance is reduced when smaller grids are used because of reducing the parallelism (thousands of threads have to run on a GPU to fully exploit its arithmetic and memory throughput). The right graph in Figure 2 depicts the scaling of the electrostatic potential evaluation. The electrostatics with the distance-dependent dielectric potential evaluation scales worse, probably because a good utilization of texture caches requires large enough problems. The computation of constant dielectrics reaches the maximum throughput when using a $100 \times 100 \times 100$ grid. The fluctuation of the results for larger grids is caused by two aspects – padding, which enforces a computation of a larger grid than is actually needed, and by utilizing such a number of blocks that is not a multiple of the count of GPU multiprocessors. The number of atoms does not affect the scaling of the electrostatic potential computation, but the memory copy overhead is more significant when less atoms are used (the size of copied grids remains the same, while the time spent on the computation is lower with less atoms).

The switch from the direct dielectrics computation to using the look-up table (see Section 4.3) is noticeable in the right graph in Figure 2, where the performance grows again around the size of the lattice 160^3.

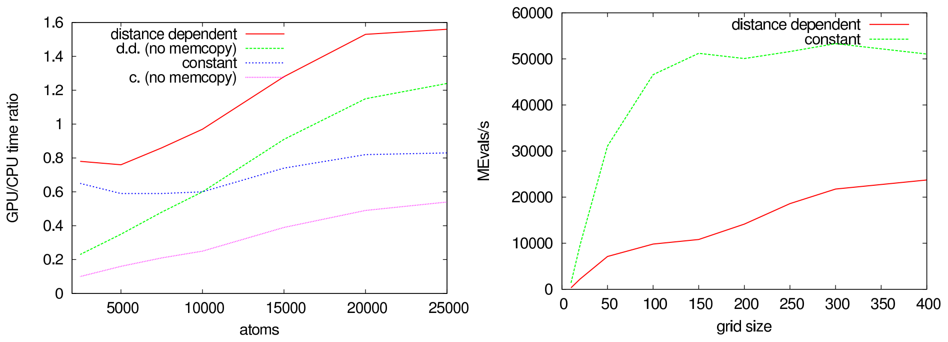

Fig. 2. Left: the ratio of GPU/CPU computation time (overall and without the CPU $\rightarrow$ GPU $\rightarrow$ CPU memory copy) for various molecule sizes; right: the scaling of the GPU implementation for various grid sizes.

Because the algorithms with worse time complexity are executed on the GPU, the speed of FastGrid is asymptotically limited by performance of the GPU. The performance of the electrostatic potential calculations with distance-dependent dielectrics is 24 391 million of atoms evaluations per second (MEvals/s) and with constant dielectrics it is 54 770 MEvals/s. The CPU code runs about 72 MEvals/s for both distance-dependent and constant dielectrics, yielding 338× and 761× speedup, respectively. However, these values are the asymptotic bound of the speedup requiring an immoderate amount of memory and computational time, thus we have performed the measurements on more commonly sized problems.

5.3 Analysis of FastGrid Arithmetic Accuracy

The today's GPUs implement the IEEE-754 standard for floating point arithmetics with a few deviations, affecting the numeric accuracy. Thus, despite the FastGrid performs equivalent computations to AutoGrid, the results vary slightly. The absolute errors (which are rounded to thousandths in output gridmap files) vary from 0 to 0.008 for all lattice points. The analysis of absolute and relative errors of FastGrid is depicted in Table 1.

Absolute error (kcal/mol)	Maximum relative error
0.001	$5.93 \cdot 10^{-5}$
0.002	$4.134 \cdot 10^{-6}$
0.003	$2.85 \cdot 10^{-6}$
0.004	$6.06 \cdot 10^{-8}$
0.005	$3.81 \cdot 10^{-8}$
0.006	$4.55 \cdot 10^{-8}$
0.007	$5.32 \cdot 10^{-8}$
0.008	$6.08 \cdot 10^{-8}$

Table 1. The difference of FastGrid results compared to AutoGrid. The table shows maximum relative error for each absolute error value that has been obtained.

The small maximum relative errors suggest that all absolute errors occur in the case of large energies, where the accuracy is not critical (the molecular docking is based on searching a minimum of energy function, thus the accuracy of high energy points is not important). Moreover, the absolute error is at most an order of magnitude higher than the smallest absolute value that can be stored in grid maps and the relative error roughly descends with the growing absolute error, thus the results are fairly accurate for all energy values.

6 CONCLUSION AND FUTURE WORK

The acceleration methods of the potential maps generation in AutoGrid yielding up to two orders of magnitude speedup have been presented. We have analyzed the potential maps generation in AutoGrid and designed the hybrid CPU/GPU parallel architecture of the new implementation called FastGrid. The efficient GPU implementation of the electrostatic potential evaluation with the novel extension allowing to compute the potential with distance-dependent dielectrics is presented.

The possible improvements of FastGrid are in the following areas:

- To get more performance, FastGrid could choose an appropriate algorithm and tweak its parameters based on the machine it is running on.
- The CPU code can be improved by a vectorization.
- The CPU/GPU load should be balanced better by using both CPU and GPU variants of one particular potential evaluation simultaneously.

Acknowledgment

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622413 and No. 0021622419 which is highly appreciated. The access to the MetaCentrum supercomputing facilities provided under the research intent MSM6383917201 is highly acknowledged.

REFERENCES

[1] Stone, J. E.—Phillips, J. C.—Freddolino, P. L.—Hardy, D. J.—Trabuco, L. G.—Schulten, K.: Accelerating Molecular Modeling Applications With Graphics Processors. Journal of Computational Chemistry, Vol. 28, 2007, No. 16, p. 2618.

[2] Morris, G. M.—Goodsell, D. S.—Halliday, R. S.—Huey, R.—Hart, W. E.—Belew, R. K.—Olson, A. J.: Automated Docking Using a Lamarckian Genetic Algorithm and an Empirical Binding Free Energy Function. Journal of Computational Chemistry, Vol. 19, 1998, p. 1639.

[3] Ufimtsev, I. S.—Martnez, T. J.: Quantum Chemistry on Graphical Processing Units; 1. Strategies for Two-Electron Integral Evaluation. Journal of Chemical Theory and Computation, Vol. 4, 2008, No. 2, p. 222.

[4] van Meel, J. A.—Arnold, A.—Frenkel, F.—Portegies Zwart, S. F.—Belleman, R. G.: Harvesting Graphics Power for MD Simulations. Molecular Simulation, Vol. 34, 2008, No. 3, p. 259, 2008.

[5] Barnes, J.—Hut, P.: A Hierarchical $\mathcal{O}(N \log N)$ Force-Calculation Algorithm. Nature, Vol. 324, 1986, pp. 446.

[6] Cheng, H.—Greengard, L.—Rokhlin, V.: A Fast Adaptive Multipole Algorithm in Three Dimensions. Journal of Computational Physics. Vol. 155, 1999, No. 2, p. 468.

[7] Hardy, D. J.—Stone, J. E.—Schulten, K.: Multilevel Summation of Electrostatic Potentials Using Graphics Processing Units. Parallel Computing, Vol. 35, 2009, No. 3, p. 164.

[8] Sousa, S. F.—Fernandes, P. A.—Ramos, M. J.: Protein-Ligand Docking: Current Status and Future Challenges. PROTEINS: Structure, Function and Bioinformatics, Vol. 65, 2006, No. 1, p. 15.

[9] Bentley, J. L.: Multidimensional Divide-And-Conquer. Communications of the ACM, Vol. 23, 1980, No. 4, p. 214.

[10] NVIDIA CUDA Programming guide, version 2.3. nVidia, 2009.

Marek Olšák has a B. Sc. degree in applied informatics from Masaryk University. Currently, he is a Master's degree student at the Faculty of Informatics, Masaryk University, and a researcher at The French National Institute for Research in Computer Science and Control. His interests include computer graphics research, soft tissue visualization, and GPU computing.

Jiří Filipovič has a M. Sc. degree in applied informatics from Masaryk University. Currently, he is a Ph. D. student at the Faculty of Informatics, Masaryk University as well as researcher in CESNET z.s.p.o. and National Centre for Biomolecular Research. His research interests include high performance computing, soft tissue modelling, haptic interactions and computational chemistry.

Martin Prokop received his Ph. D. in biomolecular chemistry from Masaryk University in 2004. He is currently a research scientist at National Centre for Biomolecular Research, Masaryk University. His research interests include development of software for computational chemistry, protein-ligand docking, modelling of enzymatic reactions, computational protein design, structural bioinformatics.