Journal of Siberian Federal University. Biology 1 (2014 7) 4-13
~ ~ ~

УДК 004.42

Calculation of Force Field Grids for Molecular Docking Using Graphics Processing Unit

Mikhail A. Farkov*
Siberian Federal University
79 Svobodny, Krasnoyarsk, 660041, Russia

Received 05.08.2013, received in revised form 13.08.2013, accepted 11.03.2014

The vast majority of problems faced by bioinformatics are very complex and time consuming. They require the use of modern high-performance computational systems and the development of algorithms for such system. Heterogeneous computing systems which include graphics processing unit (GPU) occupy a separate niche. Such systems allow to accelerate solving of some task significantly. The task performing molecular docking namely accelerating the calculation of force field grids for fast ligand-protein molecular docking is studied in this work. Algorithms for fast calculation of the large number of force field grids which are scaled to all GPUs available in the system were developed. Extensive testing on different platforms was performed.

Keywords: GPGPU, CUDA, molecular docking, virtual screening, ligand-protein docking.

Вычисление сеток силовых полей для молекулярного докинга с использованием графических процессоров

М.А. Фарков
Сибирский федеральный университет
Россия, 660041, Красноярск, пр. Свободный, 79

Подавляющее большинство задач, с которыми приходится сталкиваться биоинформатике, чрезвычайно сложны и времязатратны. Для их успешного решения требуется использовать современные высокопроизводительные вычислительные системы, для которых необходимо

* Corresponding author E-mail address: mihail.farkov@gmail.com

разрабатывать новые, специфичные алгоритмы. Среди таких систем отдельную нишу занимают гетерогенные вычислительные системы, в состав которых входят графические процессоры. Подобные системы позволяют существенно ускорить вычисления некоторых задач. В работе представлены алгоритмы, использующие возможности графических процессоров для ускорения молекулярно докинга, а именно для построения сеток силовых полей для ускорения молекулярного лиганд-белкового докинга. Предложенные алгоритмы ориентированы на быстрое вычисление большого количества сеток силовых полей и масштабируются на любое доступное компьютеру количество графических процессоров.

Ключевые слова: GPGPU, CUDA, молекулярный докинг, виртуальный скрининг, лиганд-белковый докинг.

Introduction

The rational drug design is a process that requires much time and financial investments. It takes several years to develop a new drug. The one of the key problems in rational drug design is a selection of perspective candidates in drugs. This process can take considerable period of time (1-3 years) and requires significant financial investments for chemicals and complex equipments (Kuntz, 1992). The main goal of this stage is to find some set of molecular compounds for the next stage of development, i.e. preclinical and clinical experiments. Errors at this stage of development can cause enormous losses at the next stages. Molecular docking is used to overcome the above problems. It is computer modeling of interaction of some molecules. Molecular ligand-protein docking is a modeling of interaction of a small molecule candidate in drugs (called ligand) and protein (called biotarget). Molecular docking gives information about principal possibility of reaction between molecules and gives estimates of the energy of the reaction that will be used to select perspective compounds for "in vitro" and "in vivo" experiments. Molecular docking is a quite computationally costly procedure that requires modeling of significant amounts of compounds at different spatial locations. Moreover it is necessary to estimate energy of interactions of the large number of atoms for single compounds. Thereby for such calculation it is necessary to use high performance computers that contain graphics processing units (GPU) which are very suited for solving of the proposed task.

Program for calculation of force field grids for molecular docking is presented in this paper. Graphics processing units are used in order to get speedup for the traditional grids approach. Electrostatic and van der Waals forces are taken into account.

Methods

The energy of the molecules interaction can be estimated as a linear combination of several components (Cornell et al., 1995; Jones et al., 1997; MacKerell et al., 1998).

$$E_{\text{total}} = E_{vdw} + E_{elec} + E_{bond} + E_{angle} + E_{torsion}$$

E_{vdw} is the van der Waals energy (this energy arises between atoms 2 and 5 on Fig. 1); E_{elec} is the electrostatic energy (atoms 1 and 6 on Fig. 1); E_{bond} is the energy of interaction between two covalently bound atoms (atoms 2 and 3 on Fig. 1); E_{angle} is the energy of interaction between three covalently bound atoms (atoms 1, 2 and 3 on Fig. 1); $E_{torsion}$ is the energy of interaction between covalently bound atoms that form torsion angle (atoms 1,2,3 and 4 on Fig. 1). These components can be roughly divided into

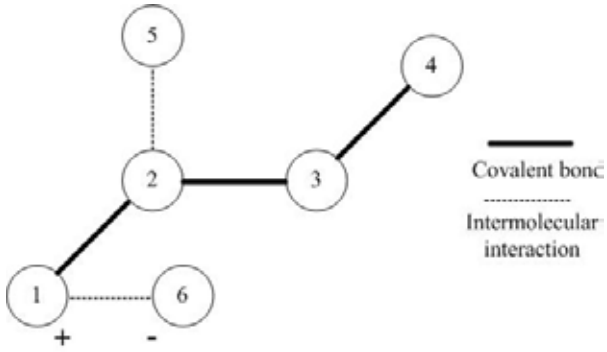

Fig. 1. Model of atoms in molecules

two groups: intermolecular and intromolecular interconnections. $E_{bond,}$ $E_{angle,}$ $E_{torsion}$ are the energies of intromolecular interconnections. These types of energies are kind of penalty that indicates that structure of molecule differs from the "ideal" structure, i.e. there are covalent bonds lengths, covalent and torsion angles that differ from the "ideal" covalent bonds lengths, covalent and torsion angles between atom types in molecule. The van der Waals and electrostatic energies are intermolecular interconnection. These types of energies are the main contributors to the total estimation of the energy. In turn, each intermolecular component can be calculated as a linear combination of interaction between each atom of ligand and each atom of biotarget.

The van der Waals energy is calculated by using the Lennard-Jones potential also known as the 6-12 potential (Goodford, 1985; Jones et al.,1997; Ewinga et al., 2001; Gilson et al., 2007):

$$E_{vdw} = \sum \varepsilon_{ij}\left(\left(r_{0ij} / r_{ij}\right)^{12} - 2\left(r_{0ij} / r_{ij}\right)^{6}\right),$$

where ε_{ij} and r_{0ij} are constants, r_{ij} is the distance between interacting atoms.

The electrostatic energy is calculated by using the Coulomb's law:

$$E_{elec} = \sum \left(q_i q_j\right) / \left(\varepsilon r_{ij}\right),$$

where ε is the permittivity, r_{ij} is the distance between interacting atoms, q_i,q_j are charges of atoms.

A common technique to accelerate molecular docking is called the grids approach. Biotarget is placed in some limited area (e.g. cuboid) and is considered as fixed. Some components of the total energy are calculated for each cell of the grid with some step for each atom type that is presented in the ligand. The probe atom type (e.g. hydrogen) is placed in each cell of the grid and the van der Waals interaction with the biotarget is calculated (Fig. 2).

This can be done only for intermolecular components of the total energy because these components don't depend on mutual location of the ligand and the biotarget. The result of the procedure is a set of grids which are used for molecular docking. The same atom types of the explored ligand will be placed at the approximately same points of the area during docking process very often. Hence during docking it is not necessary to repeat the large number of the same type of calculation for different locations of the explored ligand. One can get precalculated value of the energy from the grid for the desired atom type. The grid approach gives extra speedup for flexible ligand-protein docking since it is necessary to use additional internal degrees of freedom of the ligands for such calculation (it is

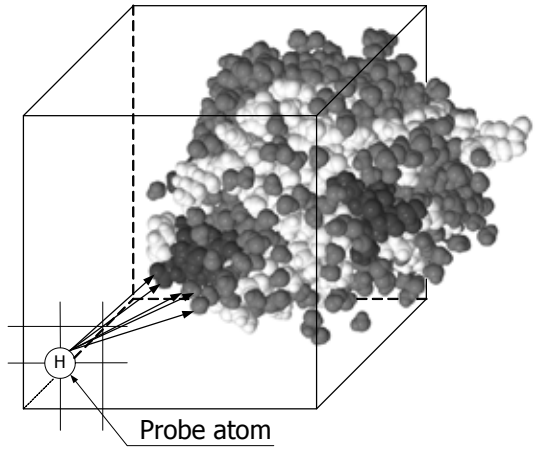

Fig. 2. Example of grid

rotation around covalent bonds and alteration of valence and torsion angles).

Implementation

There is a set of targets TARGETS at the input of the procedure where each element $target_i$ is matched to a set of ligands $LIGANDS_i$ where each element $ligand_{ij}$ is matched to a set of atom types $ATOM\_TYPES_{ij}$ such that each $atom\_type_{ijk}$ is presented in the ligand at least once. The result of the calculation is a set of grids GRIDS where each element $grid_{ijk}$ is a result of the calculation of the grid for atom type k that belongs to ligand j and interacts with biotarget i. It is true that grids for the same atom type from different ligands that interact with the same biotarget are equal, i.e.:

$$atom\_types_{ijk} = atom\_types_{lmn} \rightarrow$$
$$\rightarrow grid_{ijk} = grid_{lmn} .$$

Hence it is possible to reduce the calculation. Pair-wise calculations of the grids for each biotargets $target_i$ from the set TARGETS and each ligand from the set $LIGANDS_i$ are replaced with calculations of grid for $target_i$ and atom types that are presented in ligands from set $LIGANDS_i$ at least once.

It is not necessary to calculate grids for each charge in ligands since it is possible to evaluate the electrostatic energy at the grid's point without charges of the ligand because the biotarget is fixed and impacts equally to the points of the area and it is possible to factor out charge of ligand in the Coulomb's law. As a result it is necessary to calculate

$$\left| ATOM\_TYPES_{i0} \cup ATOM\_TYPES_{i1} \cup ... \right.$$
$$\left. ... \cup ATOM\_TYPES_{i(N-1)} \right| + 1, N := \left| LIGANDS_i \right|$$

grids for one biotarget.

Two different approaches for calculation of the permittivity for the Coulomb's law are used in this work. The constant permittivity is used in the first approach. A user sets up the necessary permittivity and this value is used during all calculation. The second approach is the distance dependent approach for evaluation of the permittivity according to the distance between point of the grid and atoms of the biotarget. The parameters offered by Mehler and Eichele are used in this work (Mehler et al., 1984; Mehler et al., 1991). This approach demonstrates significantly better results comparable with the results of the calculations using molecular dynamic. But it

takes a bit more calculation time (benchmarks are presented in the Results chapter).

Information from AMBER99 force field is used as a topology for biotargets. General AMBER force field (GAFF) is used as a resource of parameters for the van der Waals calculation. Typing the ligand molecule is performed with the Antechamber program from the AmberTools package (Wang et al., 2004; Wang et al., 2006).

Calculations of both energy components in each cell of the grid are fully independent. Tasks with such level of data parallelism are very suitable for GPU that has a lot of simple arithmetic logic units (ALU) and is oriented to SIMD programming model. Calculation of a single grid is decomposed between all ALUs of GPU in such way that a single ALU calculates energy in one cell of grid at one moment. The result of the calculation is placed in the grid that represents in GPU memory as one dimension array in such way that the X coordinate is increased most rapidly (Fig. 3). Such decomposition and memory representation guarantee that all requests to global memory will be coalesced and at least N*32 grid's point will be evaluated simultaneously where N is the number of multiprocessor in GPU. The actual number will be greater because of the different number of dispatchers on different GPU architecture.

At the same time calculation of different grids is an independent process too. It allows us to add an additional level of decomposition in case of several GPUs on one computer. The basic computational unit is a single grid which is fully calculated on the same GPU (Fig. 4). Different grids (even for one biotarget since time for reading the biotarget is short) are dispatched between different appropriate GPUs. Load balancing is performed according to the time complexity of the algorithm. The complexity for the single grid calculation is equal to $O(n*m)$ where n is the number of cells in the grid and m is the number of atoms in the biotarget. The number of atoms is approximately equal between different biotarget; is negligibly small compared with the number of cells in the grid and is unknown at the balancing stage. Hence only the number of cells is used in load balancing. Grids are dispatched in such way that a new grid is assigned to GPU with the lowest load according to the number of calculations that have been assigned to GPU.

GPU is a separate computational unit and it is necessary to provide significant load for each GPU in order to hide latency during memory exchanging. Two critical procedures in the workflow which significantly inhibit computing on GPU are reading biotargets and ligands from the disk and writing completed grids back to the disk. In order to overcome this problem and support an adequate level of load for GPU, the workflow for a single GPU is divided into three central processing unit (CPU) threads: i_thread (for input), o_thread (for output), d_thread (for calculation on GPU). All three threads share a

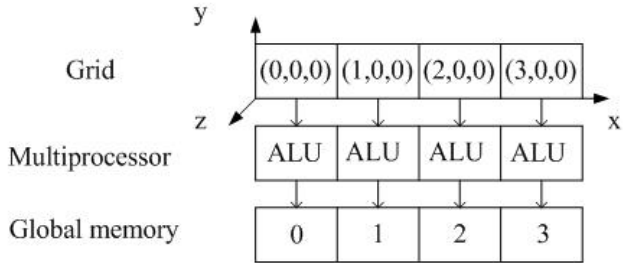

Fig. 3. Decomposition of the calculation of the single grid

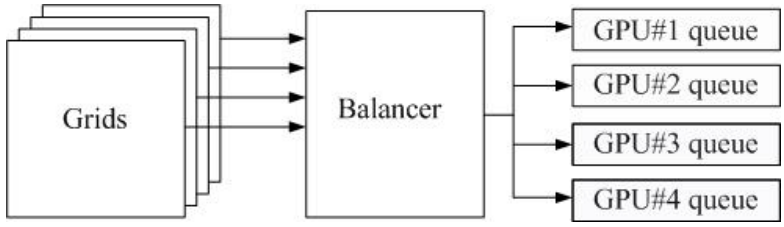

Fig. 4. Decomposition between multiple GPUs

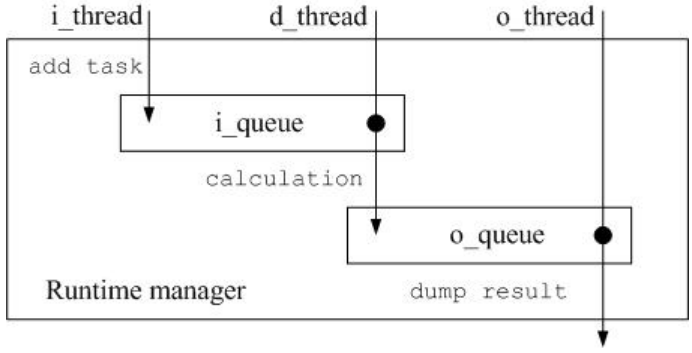

Fig. 5. Runtime manager

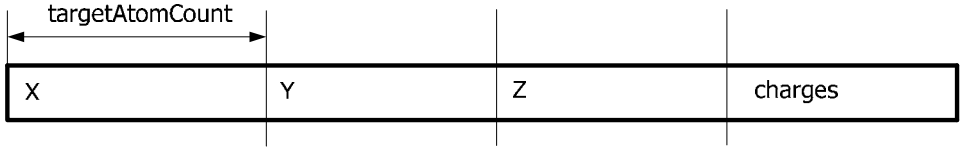

Fig. 6. The organization of information on biological target in GPU memory

special program object called a runtime manager that controls the workflow of the calculation, using of memory and errors. Moreover the runtime manager contains two queues (i_queues and o_queues) for the task for GPU (Fig. 5). There are three types of tasks (Appendix A, B): vdw (the van der Waals task), elec (electrostatic task) and elecDd (electrostatic task that uses the distance dependent approach for calculation of the permittivity). Queue for a single GPU is input of i_thread that goes through all biotargets and grids. i_thread parses biotarget, repacks it to format that is suitable for GPU, forms the runtime representations of the biotarget and puts it to a set of the runtime biotargets in the runtime manager. Atomic coordinates, charges and vdw types are necessary for the grid calculation from the biotarget. This information is repacked to one dimensional array (Fig. 6). Such organization of memory for biotarget guarantees that all GPUs threads along warps will read the exact same cell of memory hence memory requests will be coalesced.

i_thread forms the runtime representation for each new grid (independent from biotarget) and puts it to a set of runtime grids. For each ligand that is assigned to some grid i_thread runs antechamber program in order to assign

proper atom types according to GAFF. After that i_thread a adds task to i_queue according to available memory and set of atom types that are presented in ligands. The grid is calculated only for one atom type if some atom types that are presented in ligand have similar vand der Waals parameters. It is allow to reduce the calculation and at the same time save thetg flexibility of customization of the force field. If there is a low level of free memory, i_thread divides set of atom types and forms several vdw tasks. If there is no memory for a new task, i_thread waits until o_thread unloads grids to the disk. The runtime manager has a special watchdog counter to control waiting time and stops calculations if i_thread wait memory is too long. Each biotargets or grids have their own references counter which is incremented by i_thread when it adds a new task that uses this biotarget or grid. i_thread checks all references counters after each new biotarget and remove unused biotargets and grids. It is allowed to use memory more rationally and protect the system against memory leaks when the screening of the large number of biotargets/grids is performed.

d_thread gets a new task from i_queue, loads to the GPU memory runtime biotarget representation and the runtime grid representation if they have not been already loaded. After that d_thread allocates memory on GPU for grids and starts calculation according to the task type. Then d_thread dumps back the grid to CPU memory and transfers the completed task to o_queue. All steps are perform synchronously. GPU has asynchronous capabilities however they require using of CPU page-locked memory for asynchronous memory exchanging and it is not rational especially at large sets of input data and when computers with multiple GPUs are used.

o_thread gets the completed task from o_queue and dumps the completed grids to disk. After that the task is removed, all used memory is freed and references counters are decremented.

If a computer has multiple GPUs, a separate runtime manager is created for each GPU. Such separation allows to use computational resources more effectively especially on the multicore system.

Results and Discussion

All tests were performed on different modern CUDA GPUs with different computer capabilities (2.0, 2.1, 3.5) and different target markets (low-cost and HPC solutions) in order to study the behavior of algorithms on different platforms. GPUs which were used in the test are presented in Table 1.

As mentioned above two different approaches for the calculation of the permittivity were used in this work. A user can select the necessary approach depending on the required priorities (performance or accuracy). Performance tests are presented in Table 2. In general, the usual approach is faster than the distance dependent approach by 1.75-2.37 times.

The results of speedup of algorithm for GPU are presented in Table 3. The base value

Table 1. Test GPUs

GPU	Computer capabilities	Multiprocessors	ALU	Market
Tesla M2090	2.0	16	512	HPC
Tesla K20	3.5	13	2496	HPC
GeForce GT 440	2.1	2	96	Low-cost

for comparison is the result of the calculation of 9 grids (size along all dimensions is 64) for one biotarget with using the program AutoGrid4 that was developed in Scripps Research Institute and is used for grids calculation for the program AutoDock4 (Morris et al., 1997; Morris et al., 2009). The tests were performed on the computer with Intel Core i7 920 and 3 GB RAM. Speedup varies on different GPUs but even on a low-cost GPU with the low number of ALUs it is significant.

The results of speedup of implementation with the runtime manager and with different threads for input, output and execution (Fig. 6) are presented in Table 4. The tests were performed by using two Tesla M2090. They included three cases with 60 grids with different sizes and different ligands for each grid. This conception gives additional speedup regardless of grids size and hides latency of memory exchange.

Additional tests in order to estimate quality of load balancing were performed. Two arrays of GPU were used: 2 GPU Tesla M2090 and 4 GPU Tesla K20. It is a common configuration of arrays of GPU. The average time of execution of task on each GPU and unloading completed grid to the disk for a pair of Tesla M2090 are presented in Table 5.

Similar tests were performed for a quartet of Tesla K20. The results are presented in Table 6. Load is distributed well across multiple GPUs

Table 2. The average time of the kernel with (dd column) and without (not dd column) using the distance dependent approach

GPU	not dd,ms	dd,ms	Speedup (dd/not dd)
Tesla M2090	126.15	220.44	1.75
Tesla K20	87.25	205.82	2.36
GeForce GT 440	740.295	1758.09	2.37

Table 3. Average time of calculation of 9 grids using different GPUs

GPU	Elapsed,ms	Speedup
AutoGrid 4	74010	1
Tesla M2090	1824.04	40.58
Tesla K20	1523.21	48.59
GeForce GT 440	11370.01	6.51

Table 4. Results of comparison between implementation using single thread and implementation using separate threads (i_o_d column)

Test case	single thread,s	i_o_d,s	speedup
64*64*64	221	145	1.52
128*128*128	1296	877	1.48
Varying size 32-128	702	334	2.11

Table 5. Execution time (d_thread) and dumping time (o_thread) on 2 Tesla M2090

Test case	d_thread,s		o_thread,s	
	GPU #0	GPU #1	GPU #0	GPU #1
64*64*64	124	128	114	117
128*128*128	761	754	620	585
Varying size 32-128	235	237	290	291

Table 6. Execution time (d_thread) and dumping time (o_thread) on 4 Tesla K20

Test case	d_thread,s				o_thread,s			
	GPU #0	GPU #1	GPU #2	GPU #3	GPU #0	GPU #1	GPU #2	GPU #3
64*64*64	41	43	45	46	69	73	74	76
128*128*128	264	256	267	255	459	421	470	442
Varying size 32-128	96	86	105	95	165	136	163	159

independent from size of array without idle of GPU.

The set of test results shows that the proposed algorithms are computationally effective. They use maximum of computer and GPU resources and scale for different platforms well. The algorithms are implemented in a program for very fast calculation of the large number of grids of force fields GPU-CGFF.

The program is registered in the Federal Institute of Industrial Property of Russian Federation on November 14, 2013. Registration number is 2013660687.

Conclusion

Proposed approach for force field grids calculation demonstrates high performance compared with traditional CPU approaches. Algorithm efficiently and automatically scales computational load on multiple graphics processors. Commonly used force field and accurate distance dependent approach for evaluation of the permittivity are used. The algorithms are useful for acceleration of the ligand-protein molecular docking.

Acknowledgment

The author thanks Sergey Feranchuk from United Institute of Informatics Problems of the National Academy of Sciences of Belarus for helpful consultations during this work; The HPC Research Department of Siberian Federal University and The Tauber Bioinformatics Research Center at University of Haifa for providing of computing resources that were used to carry out the research.

References

1. Brooijmans N., Kuntz I.D. (2003) Molecular recognition and docking algorithms. Annual Review of Biophysics and Biomolecular Structure 32: 335-373.
2. Cornell W.D., Cieplak P., Bayly C.I., Gould I.R., Merz K.M., Ferguson D.M., Spellmeyer D.C., Fox T., Caldwell J.W., Kollman P.A. (1995) A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. Journal of the American Chemical Society 117 (19): 5179-5197.

3. Ewing T.J.A., Makino S., Skillman A.G., Kuntz I.D. (2001) DOCK 4.0: Search strategies for automated molecular docking of flexible molecule databases. Journal of Computer-Aided Molecular Design 15: 411-428.
4. Gilson M.K., Zhou H.X. (2007) Calculation of protein-ligand binding affinities. Annual Review of Biophysics and Biomolecular Structure 36: 21-42.
5. Goodford P.J. (1985) A computational procedure for determining energetically favorable binding sites on biologically important macromolecules. Journal of Medicinal Chemistry 28: 849-857.
6. Jones G., Willett P., Glen R.C., Leach A.R., Taylor T. (1997) Development and validation of a genetic algorithm for flexible docking. Journal of Molecular Biology 267: 727-748.
7. Kuntz I.D. (1992) Structure-based strategies for drug design and discovery. Science 257 (5073): 1078-1082.
8. MacKerell A.D., Bashford D., Dunbrack R.L.et al. (1998) All-atom empirical potential for molecular modeling and dynamics studies of proteins. The Journal of Physical Chemistry 102 (18): 3586-3616.
9. Mehler E.L., Eichele G. (1984) Electrostatic effects in water-accessible regions of proteins. Biochemistry 23 (17): 3887-3891.
10. Mehler E.L., Solmajer T. (1991) Electrostatic effects in proteins: comparison of dielectric and charge models. Protein Engineering 4(8): 903-910.
11. Morris G.M., Goodsell D.S., Halliday R.S., Huey R., Hart W.E., Belew, R.K., Olson A.J. (1998) Automated docking using a Lamarckian genetic algorithm and empirical binding free energy function. Journal of Computational Chemistry 19: 1639-1662.
12. Morris, G.M., Huey, R., Lindstrom, W., Sanner, M.F., Belew, R.K., Goodsell, D.S., Olson, A.J. (2009) AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility. Journal of Computational Chemistry 30: 2785-2791.
13. Wang, J., Wolf R.M., Caldwell J.W., Kollman P.A., Case D.A. (2004) Development and testing of a general AMBER force field. Journal of Computational Chemistry 25: 1157-1174.
14. Wang, J., Wang, W., Kollman, P.A., Case D.A. (2006). Automatic atom type and bond type perception in molecular mechanical calculations. Journal of molecular graphics modeling 25 (2): 247-260.

Appendix A. CUDA kernel for van der Waals calculation

```
__global__ void vdwMultipleKernel(float *deviceTargetInfo,float *deviceRadius,float
*deviceEpsilon,float *deviceProbE,float *deviceProbR,unsigned short int
*deviceProbCount,unsigned char *deviceTypes,unsigned int *deviceProteinAtomCount,float
*deviceVdwGrid,float *deviceGridParameters,unsigned short int *deviceGridSize)
{
  unsigned short int registerProbCount=*deviceProbCount, xIndex,yIndex,zIndex;
  unsigned int registerProteinAtomCount=*deviceProteinAtomCount,
    numberOfCellInGrid=deviceGridSize[0]*deviceGridSize[1]*deviceGridSize[2];
  float currentX,currentY,currentZ, r,e,r0;
  //Loop over all cells
  for(unsigned int index=blockIdx.x*blockDim.x+threadIdx.x;  index<numberOfCellInGrid;
    index+=gridDim.x*blockDim.x){

    for(unsigned short int probIndex=0;probIndex<registerProbCount;++probIndex)
      deviceVdwGrid[numberOfCellInGrid*probIndex+index]=0;

    xIndex=index/(deviceGridSize[1]*deviceGridSize[2]);
    yIndex=(index%(deviceGridSize[1]*deviceGridSize[2]))/deviceGridSize[2];
    zIndex=index%deviceGridSize[2];
    currentX=xIndex*deviceGridParameters[3]+deviceGridParameters[0];
    currentY=yIndex*deviceGridParameters[3]+deviceGridParameters[1];
    currentZ=zIndex*deviceGridParameters[3]+deviceGridParameters[2];

    //Loop over target atoms
    for(unsigned int atomsIndex=0; atomsIndex<registerProteinAtomCount;++atomsIndex){
      r=__fsqrt_rn((deviceTargetInfo[atomsIndex]-currentX)*
        (deviceTargetInfo[atomsIndex]-currentX)+
        (deviceTargetInfo[atomsIndex+registerProteinAtomCount] – currentY)*
        (deviceTargetInfo[atomsIndex+registerProteinAtomCount]-currentY)+
        (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2] -currentZ)*
        (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]-currentZ));
      //Loop over probes
      for(unsigned short int probIndex=0;probIndex<registerProbCount;++probIndex){
        r0=deviceProbR[probIndex]+deviceRadius[deviceTypes[atomsIndex]];
        e=__fsqrt_rn(deviceProbE[probIndex]*deviceEpsilon[deviceTypes[atomsIndex]]);
        deviceVdwGrid[numberOfCellInGrid*probIndex+index]+=
          e*(__powf((r0/r),12)-2*__powf((r0/r),6))*((unsigned short int)
          (deviceTargetInfo[atomsIndex]>=deviceGridParameters[0]&&
          deviceTargetInfo[atomsIndex]<=
          (deviceGridParameters[0]+deviceGridParameters[3]*deviceGridSize[0]))&&
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount]>=
          deviceGridParameters[1]&&
          deviceTargetInfo[atomsIndex+registerProteinAtomCount]<=
          (deviceGridParameters[1]+deviceGridParameters[3]*deviceGridSize[1]))&&
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]>=
          deviceGridParameters[2]&&
          deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]<=
          (deviceGridParameters[2]+deviceGridParameters[3]*deviceGridSize[2])));
}}}}
```

Appendix B. CUDA kernel for electrostatic calculation.

```
__global__ void elDdPrototypeKernel(float *deviceTargetInfo,unsigned int
*deviceProteinAtomCount,float *deviceCoulombPrototype,float *deviceGridParameters,unsigned
short int *deviceGridSize,float *devicePermittivity)
{
   unsigned int registerProteinAtomCount=*deviceProteinAtomCount,
      numberOfCellInGrid=deviceGridSize[0]*deviceGridSize[1]*deviceGridSize[2];
   unsigned short int inGrid, xIndex,yIndex,zIndex;
   float coulomb,r,currentX,currentY,currentZ;

   for(unsigned int index=blockIdx.x*blockDim.x+threadIdx.x;
      index<numberOfCellInGrid;index+=gridDim.x*blockDim.x){
        xIndex=index/(deviceGridSize[1]*deviceGridSize[2]);
        yIndex=(index%(deviceGridSize[1]*deviceGridSize[2]))/deviceGridSize[2];
        zIndex=index%deviceGridSize[2];
        currentX=xIndex*deviceGridParameters[3]+deviceGridParameters[0];
        currentY=yIndex*deviceGridParameters[3]+deviceGridParameters[1];
        currentZ=zIndex*deviceGridParameters[3]+deviceGridParameters[2];

      coulomb=0;
      for(unsigned int atomsIndex=0;atomsIndex<registerProteinAtomCount;++atomsIndex){
          r=__fsqrt_rn((deviceTargetInfo[atomsIndex]-currentX)*
          (deviceTargetInfo[atomsIndex]-currentX)+
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount]-currentY)*
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount]-currentY)+
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]-currentZ)*
          (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]-currentZ));
          inGrid=((unsigned short int)
            (deviceTargetInfo[atomsIndex]>=deviceGridParameters[0]&&
            deviceTargetInfo[atomsIndex]<=
            (deviceGridParameters[0]+deviceGridParameters[3]*deviceGridSize[0]))&&
            (deviceTargetInfo[atomsIndex+registerProteinAtomCount]>=
            deviceGridParameters[1]&&
            deviceTargetInfo[atomsIndex+registerProteinAtomCount]<=
            (deviceGridParameters[1]+deviceGridParameters[3]*deviceGridSize[1]))&&
            (deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]>=
            deviceGridParameters[2]&&
            deviceTargetInfo[atomsIndex+registerProteinAtomCount*2]<=
            (deviceGridParameters[2]+deviceGridParameters[3]*deviceGridSize[2])));

          coulomb+=(deviceTargetInfo[atomsIndex+registerProteinAtomCount*3]/(r*r))*
            inGrid*(ELEC_DD_A+(*devicePermittivity-ELEC_DD_A)/
            (1+ELEC_DD_K*__expf(ELEC_DD_L*(*devicePermittivity-ELEC_DD_A)*r)));
      }
   deviceCoulombPrototype[index]=coulomb;
}}
```