Accelerating Protein-Protein Docking using a Graphics Processing Unit (GPU)

Michael Jenik[1], Esteban Mocskos[1,2], Adrián E. Roitberg[3] and Adrián G. Turjanski[1⋆]

[1] Departamento de Química Inorgánica, Analítica y Química Física; Instituto de Química Física de los Materiales, Medio Ambiente y Agua and Departamento de Química Biológica, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires, Argentina

[2] Laboratorio de Sistemas Complejos, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires
Buenos Aires (C1428EGA), Argentina

[3] Department of Physics, University of Florida, Gainesville, Florida 32611-8435, USA.

Abstract. Solving the structure of protein-protein complexes is one of the most important tasks in structural biology. Even though there has been great progress in recent years there still a small number of protein complexes structures deposited in the Protein Data Bank in comparison to isolated partners. In this sense, the computational prediction of protein complexes starting from the unbound structures, protein-protein Docking algorithms, has emerged as a reasonable alternative. Many docking programs employ Fast Fourier Transform (FFT) correlations as an efficient search strategy. We describe an implementation of a protein-protein docking program based on FFT surface complementarity that runs entirely on a Graphics Processing Unit (GPU), including grid generation, rotation and scoring. We evaluate its performance, and show that it can be up to 13 times faster than conventional CPU based implementations.

1 Introduction

As protein-protein interactions play a central role in almost any physiological process, the structural characterization of protein-protein complexes is key to understand normal and pathological cell function [1]. Knowledge of the protein complex structures can shed light into the functions of the component proteins and can guide the design of novel drugs to regulate the complex protein interaction networks [2].Even though there are more than 50000 protein structures deposited up to date in the protein data bank [3] only a small subset correspond to protein complexes as the experimental determination of their 3D

⋆ Author to whom correspondence should be addressed at: Laboratorio de Bioinformática Estructural, Departamento de Química Biológica, Ciudad Universitaria, Pabellón II, Facultad de Ciencias Exactas y Naturales, UBA, Buenos Aires, Argentina. E-mail: `adrian@qi.fcen.uba.ar` structures has remained difficult. In this sense, there is a pressing need to develop reliable and rapid computational methods for predicting protein-protein complexes structures at a genomic scale [4]. Docking algorithms attempt to predict the structure of complexes formed by two or more interacting biological macromolecules starting from the structure of the isolated partners [5, 6].

There has been a wealth of research on protein-protein docking and every year results from blind docking experiments show that the performance of the algorithms is improving in efficiency, reliability and accuracy [7]. However, in most cases current algorithms still have difficulty in identifying the correct solution.

The majority of the docking programs contain a search algorithm that samples possible docking orientations efficiently and a scoring function that aims to discriminate near native docked orientations from incorrect ones. Many of these approaches can be computationally intensive as the search space consists of all possible orientations and conformations of the protein paired with the ligand. Shape complementarity is the most basic ingredient of the scoring functions for docking as it is known that protein surfaces are complementary to each other at the binding interface. One way of calculating the surface complementarity is to discretize each protein in a grid were each voxel has a different value depending if it lies inside, in the surface or outside the protein. One protein, usually the biggest one, is consider the receptor and remains fixed. The other one, the ligand, is translated and rotated accounting for all possible positions. The total number of surface voxels in the receptor that overlap any ligand surface voxel (which approximates buried surface area upon complexation), minus a penalty due to the overlapping of voxels that lies inside the two proteins, is proportional to the surface complementarity score. Katchalski-Katzir et al developed a FFT-based algorithm that was able to explore all possible translational orientations rapidly while computing the surfaces complementarity at the same time [8]. The computational speed up comes because FFT allows a problem that formally requires $O(N^2)$ operations to be computed in $O(N \log N)$ steps [8]. This method has then been adopted and extended in later works by other groups, like the programs ZDOCK [9], FTDock [10], 3D-Dock [11], GRAMM [12,13] and DOT [14]. Several groups have developed multiterm interaction potentials and others use multi-copy approaches to simulate protein flexibility, which both add to the computational cost of FFT-based docking algorithms [10, 14, 9, 15, 16]. In this sense there is a need to develop more efficient FFT docking techniques.

Attention have been attracted to Graphics Processing Unit (GPU) as high-performance computing devices. GPU computation has been successfully applied to various types of applications, including molecular dynamics, quantum chemistry, and quantum Monte Carlo. We present an implementation of a FFT-docking algorithm that fully runs in the GPU, including grid generation, rotation and scoring, that reduced the computational time up to an order of magnitude than that of latest commodity CPUs. We show that the different stages of the docking process must be run in the GPU to efficiently reduce the computational cost. This approach is applied to a benchmark of complexes previously studied using shape-only correlations. The implementation presented here can be easily incorporated into any docking program based on FFT and can be easily extended to more complex scoring functions.

2 Materials and Methods

The increasing demand for sophisticated graphics for video games, computeraided design (CAD), animation, and other applications is driving the development of more and more powerful graphical processing units, which take advantage of data parallelism to render graphics at high speeds. The recent release of graphics card manufacturer NVIDIA's Compute Unified Device Architecture (CUDA) development toolkit for some of their high-end graphics cards allows developers to code algorithms in a C-like language. CUDA greatly eases the transition from using CPUs to general-purpose computing on GPUs (GPGPU). Graphical processors are able to outperform CPUs for certain applications because of intrinsic parallelization within the device.

Multicore and parallel CPU architectures, though able to run many instructions simultaneously, require computational threads to be explicitly coded to make optimal use of the available resources. Whereas a single-core CPU can only execute a single instruction at a time (although several instructions may be in the pipeline), a GPU can execute a single instruction on many pieces of data at the same time, using a Single Instruction, Multiple Data (SIMD) paradigm. This inherent parallelization is a result of hardware architecture; graphics cards are composed of an array of multiprocessors, each of which has its own section of pipeline bandwidth.

Data access from the internal math units to GPU local memory is slow compared to computation, but transferring data between the GPU and CPU across the PCIe bus is still much slower. For this reason, communication between the GPU and CPU should be kept to an absolute minimum. Ideally, the simulation should be executed entirely on the GPU, and results should be sent back to the CPU only infrequently for analysis and reporting.

Jamaica has been implemented both in *Python* calling pure ANSI C functions using FFTW3 library [17] for Fourier transformation computation, and in *Python* calling CUDA-C [18] functions which uses the GPU.

In this work, two different computational setups were used:

1. 2 dual core Opteron CPUs (2.2GHz, 1MB cache), 4 GB of RAM. Operating System Red Hat 4.1.2-42, kernel version 2.6.18-92.el5 and Tesla C1060 card. It has 240 streaming processor cores working at 1.3GHz and 4GB DDR3 of dedicated memory.
2. 1 Pentium dual core CPU (2.2GHz, 1MB cache), 2 GB of RAM, motherboard ASUS P5KPL-CM. Operating system Ubuntu 8.10, kernel version 2.6.27-7-generic and GeForce 8800 GTX board. This card manufactured by PYN has 128 streaming processor cores working at 575MHz and 768MB DDR3 of dedicated memory.

The figure 1 shows the main program structure and the data flow. The arrows represent the information flow between the different software modules. The big box on the right titled GPU contains the Jamaica main operations that can be selected to be computed in CPU or GPU.

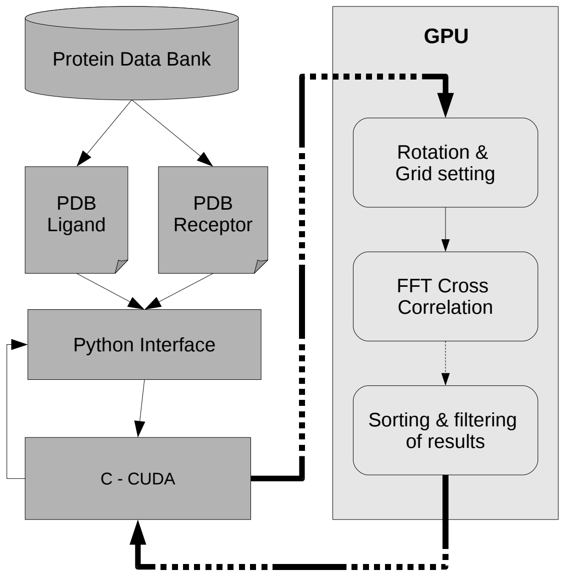

Fig. 1. Modules composing the Jamaica docking program. The rounded part on the right indicates the functions that are designed to support computation in the CPU or GPU.

A brief description of the module's functions is given assuming the GPU is selected to be used for the computation:

1. Jamaica needs the crystallographic structures of ligand and receptor which are stored in PDB files. The docking procedure can be applied to the subunits extracted from the entire crystallized complex (using only one PDB file) or to two different units (using two PDB files, one for the ligand and one for the receptor).
   So, the first action is to read the Protein Data Bank (PDB) files and calculate the size of the boxes containing the two subunits to dock. Each file contains the position of the atoms forming the protein. This is implemented with *Python* and computed in the CPU. Then the positions are transferred to the GPU main memory.
2. The receptor is rendered and its FFT is calculated in the GPU (all the invocations are stated using the *Python* interface to *CUDA API*). The rendering procedure consists in asigning a value to each position in a discretized 3D grid, which represents the space. The rule for asigning values to the grid nodes depends on the positions of each node and molecule' atoms:

   a) C is used for the core if the node distance from molecule's atoms is less than an user-defined threshold D, normally selected in the order of half van der Waals (VDW) distance.
   b) S is selected if the node distance from molecule's atoms is more than D and less than a $D + T$, where T is the surface thickness, usually in the order of half VDW radius.

   S and C are complex values chosen to achieve the next relations:
   i) a high positive value of $S \times S$ represents a surface sharing between the ligand and receptor.
   ii) a value near zero of $S \times C$ represents the one of the molecules is getting near the other core. This is a very usual behaviour in rigid docking.
   iii) a high negative value of the real part of $C \times C$ results in a core superposition. This behaviour is not desired, and a penalty must be applied.
3. Using the rotation step selected by the user, the rotation matrices are calculated and stored in disk for later use.
4. One of the unused rotation matrices is transferred to the GPU.
5. Rotation & Grid setting: includes the rotation and rendering of the receptor in GPU.
6. FFT for the ligand is performed in the GPU.
7. Value to value multiplication of ligand and receptor is performed in GPU. The result of this procedure is stored in the memory area of the ligand to save space. This grid is now called *multiplication grid*. This is part of the cross correlation computation.
8. FFT for the multiplication grid is performed (in GPU using *CUFFT*). This is part of the cross correlation computation.
9. sorting and filtering of the intermediate results are performed in GPU and then transferred to CPU (results sorting & filtering).
10. The results obtained are merged with the previous ones, then the whole set is sorted. Only the first 1000 results are kept, the others are discarded (this value can be configured by the user). These operations are done in CPU using *Python*. A cycle is established to step 4 until no more rotation matrices remain unused.
11. Final results are written to disk.

The steps 1 to 3 constitute the initial serial start-up. A cycle is established between steps 4 and 10 repeating the computation using all the pre-calculated rotation matrices.

In the following, we give some implementation details of the GPU based modules:

Grid Generation: One of the most time consuming operation is computing the FFT, which is implemented using CUDA FFT library (CUFFT). The other time consuming operation is the the renderization of the rotated protein. As was mentioned before, the molecule is composed of an inner core space and a surface volume wrapping it. The rendering procedure consists in initially resetting all the points inside the box containing the molecule, then the points occupied by the molecule are *painted* with the value S. Finally, the

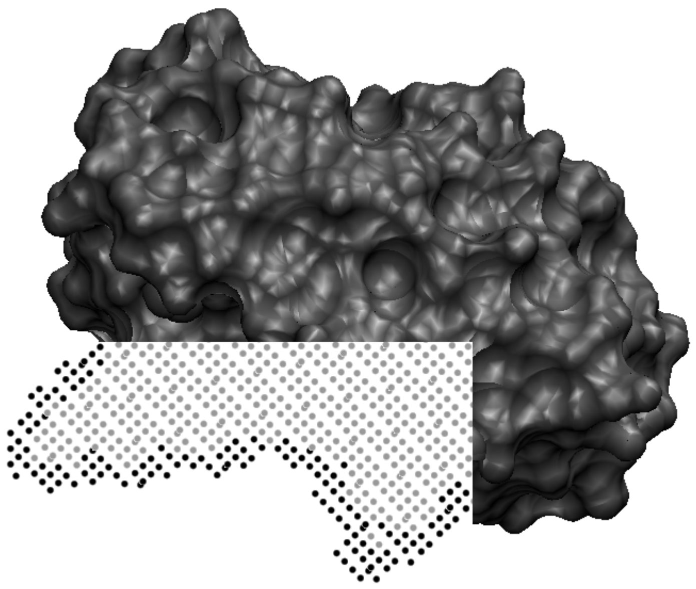

Fig. 2. Renderized grid of the protein included in the 1AVX.pdb file. The external view of the molecule surface is superimposed with a slice showing the assigned values for each atom, the light grey dots represent the molecule core and black dots indicate the surface.

core is determined and *painted* using the value C. The first step in the *painting* procedure is to determine the smallest *box* containing each atom. Then the points corresponding to the sphere that represents the atom inside the box are assigned C or S value according to the position. The points inside the box but outside the sphere remain unchanged.
In the CUDA implementation, a block of threads is assigned for painting each sphere. A single thread is used to compute the value to be used in each point of the corresponding box. If the amount of points in the box overpasses the maximum number of threads per block of the used GPU, then a cycle is established to process the remaining points. The code listing 1.1 shows the kernel source code for the sphere painting function. Some interesting details of this code are given:

– Due to the fact that the same program must be executed in all threads simultaneously, the same set of parameters are received and each thread must obtain the assigned subset of input values. In this routine, two arrays are received: one with the information regarding the previously calculted *color* (i.e. core, surface) to use in the painting of the sphere containde in the box and one with the coordinates of each sphere's center.
– In the line 3, each block of threads obtains the index of the sphere to paint, i.e. this step corresponds to the job assignment.

– Using the index obtained in the step 2, the lines 4, 5 and 6 obtain from the received arrays, the coordinates of the sphere center, the sphere radious and the complex value to use for the painting (previously obtained in another part of the program).
– In the line 13, a cycle is established for the case in which the number of points to paint is larger than de number of threads per block, this parameter depends on the hardware used.
– A coordinate transformation is done in the line 15 from a system based on one of the box corners to a system with origin in the center of the box. In particular, each thread computes its own coordinate using the thread-id array (named `threadIdx`).
– The line 17 corresponds to the transformation of the sphere's center in floating point to integer. The idea behind this is to have a point which represents the center of the sphere in global coordinates.
– In the line 19 a vector is computed. This vector joins the point assigned to the thread and the sphere's center, the components of the obtained vector are expressed in global coordinate system. The difference between the sphere's center and the coordinates obtained in the previous steps is used (after converting those coordinates to Ångströms).
– In the line 21, the distance from the sphere's center to the assigned point is calculated.
– Then, in the line 22, the algorithm checks if the point is inside the sphere or not. If it is inside, then the painting value is stored in the GPU memory indexing in the array which contains all the points.
– Finally, the lines 23 and 24 obtain the index of the array corresponding to the point to be painted.

A renderized grid of the protein included in the 1AVX.pdb file (corresponding to the Soybean Trypsin Inhibitor, Tetragonal Crystal Form) is shown in the figure 2, which contains a visualization of the external view of the molecule surface superimposed with a slice showing the assigned values for each sphere, the light grey dots represent the molecule core and black dots indicate the surface.

Listing 1.1. Kernel code for rendering an atom represented by one sphere. The value used for the *painting* procedure is one of the function parameters.

```
1   __global__ void _paintSphere(cuComplex *_vol, dim3 _shape, float3
        _delta, float3 _resol, float3 *v_x, float *v_r, cuComplex *v_v,
        int _n, float4 *_T, int offsetSphere , int wishedDimZ) {
2
3     const uint i = offsetSphere + blockIdx.x * gridDim.y * gridDim.z +
          blockIdx.y * gridDim.z + blockIdx.z;
4     float3 _x_=v_x[i], _radiecito_;
5     const float _r=v_r[i], __radio2 = _r*_r;
6     const cuComplex _v = v_v[i];
7     const float3 __resol2 = _resol*_resol;
8     int3 idx, c, p;
9     int ref;
10    float sum;
11    uint offsetThreadIdx_z;
12
13    for( offsetThreadIdx_z=0; offsetThreadIdx_z < wishedDimZ;
          offsetThreadIdx_z += blockDim.z ){
```

```

        p = make_int3( threadIdx.x - (blockDim.x-1)/2, threadIdx.y - (
            blockDim.y-1)/2, threadIdx.z - (wishedDimZ-1)/2 +
            offsetThreadIdx_z );

        c = make_int3( floorf(_x_.x/_resol.x), floorf(_x_.y/_resol.y),
            floorf(_x_.z/_resol.z) );

        _radiecito_ = tofloat3(p+c)*_resol - _x_;

        sum = len2( _radiecito_ );
        if ( sum <= __radio2 ){
            idx = p + c;
            ref = reference( idx , _shape );
            _vol[ref] = _v;
        }
    }
}
```

Sorting and Filtering: After every cross correlation computation, the obtained grid contains a score value in each position. These values must be sorted and filtered to keep only a subset of them. In GPU implementation two algorithms were developed based on the ideas behind CUDA Data Parallel Primitives (CUDPP) [19]. A modified version of Radix Sort was implemented to obtain not only the data vector but also a permutation vector. On the other hand, a fast threshold filtering of scores was implemented based on the CUDPP `compact` function. The original function takes as input a data vector and a logical vector which indicates if each element of the first vector must be included in the *compacted* output or not. In this case, the function was modified to operate at the same time in two data vectors: the data and permutation vectors. A threshold comparation routine was also implemented to generate the logical vector needed by the modified `compact` function. After computing each rotation, only the best 1000 solutions are transfered to the CPU main memory to be compared with the best solutions of the previous rotations.
The CPU implementation of this module use Heap sort and then applies a cut-off procedure to the sorted vector in order to keep the top 1000 solutions.

The figure 3 shows the use of computing time of Jamaica main modules. As was mentioned before, the FFT related operations are the top processor cicle consumers followed by the grid generation procedure and, finally, by the sorting & filtering process. Nevertheless, the FFT related operations constitute almost the 80% of time in the case of GPU and 90% for the CPU based solution.

3 Results and Discussion

Human hemoglobin (HH) consists of four subunits: two pairs A, and two pairs B. It is a very good target for testing a docking procedure because it has been crystallized not only the entire complex but also each subunit separately. The differences between the subunit structures obtained in these two cases are minimal. Due to fact that the deformation produced in the docking process is so small that HH turns to be an excellent testing bed specifically for rigid docking [8].

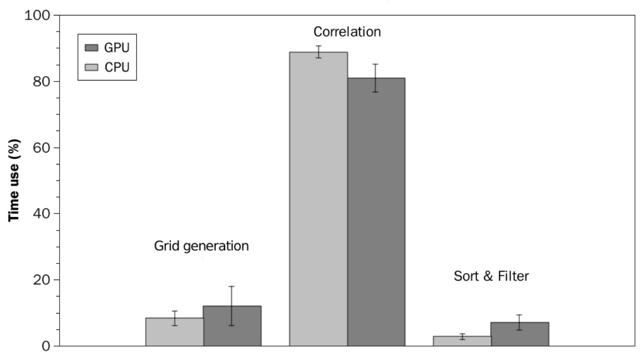

Fig. 3. Computing time distribution of the main Jamaica modules

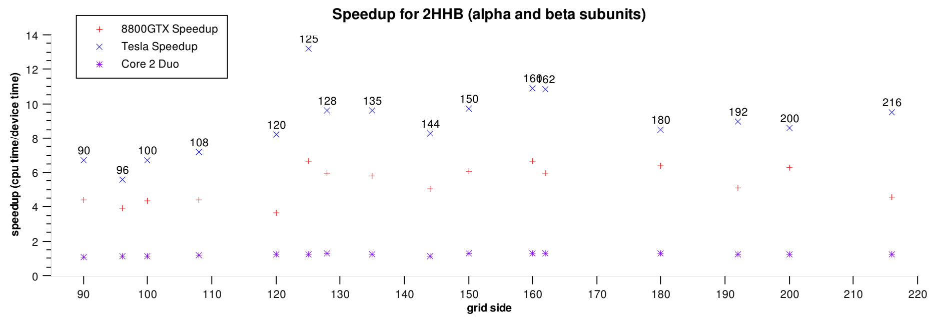

Fig. 4. Speedup obtained for the Human hemoglobin (HH) using selected grid size values in different computational setups. For grid sizes larger than 125^3 nodes the obtained speedup for the version using a 8800 GTX board is at least 5X. With the Tesla board, the speedup for grid size larger than 120^3 nodes is at least 8X.

The testing procedure for Jamaica consists in the docking of the HH subunits A and B included in the `2HHB.pdb`, which includes the structure for the bounded HH complex. The methodology involves the evaluation of the rotational space uniformely distributed using 10 degrees rotational steps, resulting in 17.497 single rotations.

The Cooley-Tukey algorithm implemented using FFTW3 and CUFFT share the feature of being very efficient when the factorization of the dataset dimention is composed of low values. For this reason, the grid size was generated following $2^a * 3^b * 5^c$, with a , b , and c in the range from 0 to 7; leading to cubic grids of 75^3 (1.406Å resolution), 81^3 , 90^3, . . . 200^3, 216^3 (0.488Å resolution) nodes.

The figure 4 shows the speedup obtained for Jamaica in each selected grid size. The base time used for the speedup calculation corresponds to the CPU version of Jamaica executed in a machine with a Core 2 Duo processor with only one core enabled. This figure includes the result for three computational setups: a CPU version with a Core 2 Duo processor using the two cores, a GPU version using the 8800 GTX board and a GPU version with a Tesla C1060 board. The results obtained with the 8800 GTX board show up to 6.5 times speedup, for grid sizes larger than 125^3 nodes the obtained speedup is at least 5X. With the Tesla board, the speedup for grid size larger than 120^3 nodes is at least 8X, reaching 13X for the case of 125^3 nodes. The results obtained using the two cores of the CPU do not show significant improvement compared with the single core case.

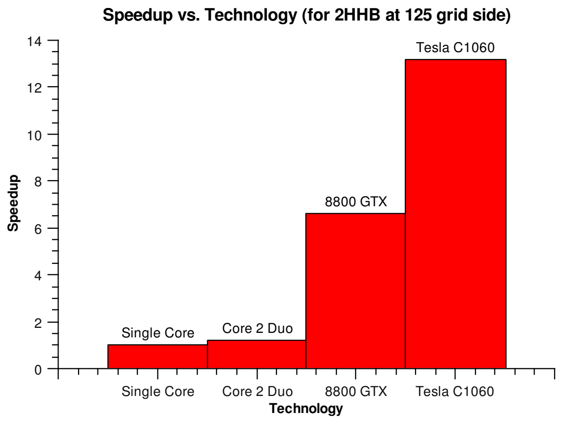

Fig. 5. Grid containing 125^3 nodes, the CPU version with two cores shows almost no gain compared with the single core version. The 8800GTX version shows a 6.5X speedup, while the Tesla version reaches 13X.

In the figure 5 the case with the grid containing 125^3 nodes was selected to be analysed. The column named "Single Core" corresponds to the base case used in the previous analisys for computing the speedup. In this case, the 8800 GTX version achieves a speedup near 6.5X, while the Tesla version reaches 13.5X overmatching the others technologies evaluated. The single and dual cores CPU versions maintain the same behavior previously described.

4 Conclusions

Solving the structure of protein-protein complexes is one of the most important tasks in structural biology. Even though there has been great progress in recent years there still a small number of protein complexes structures deposited in the Protein Data Bank in comparison to isolated partners. In this sense, the computational prediction of protein complexes starting from the unbound structures, protein-protein Docking algorithms, has emerged as a reasonable alternative.

Multicore and parallel CPU architectures, though able to run many instructions simultaneously, require computational threads to be explicitly coded to make optimal use of the available resources. Graphics cards are composed of an array of multiprocessors, each of which has its own section of pipeline bandwidth. Data access from the internal math units to GPU local memory is slow compared to computation, but transferring data between the GPU and CPU across the PCIe bus is still much slower, leading to the necessity of keeping the communication between the GPU and CPU to an absolute minimum.

Jamaica has been implemented both in *Python* calling ANSI C using FFTW3 library, and in *Python* calling CUDA-C functions which uses the GPU and was tested using the Human hemoglobin (HH). The results obtained with the 8800 GTX board show up to 6.5 times speedup. With the Tesla board, the speedup reached 13X.

Acronyms

CUDA	Compute Unified Device Architecture	3
CUDPP	CUDA Data Parallel Primitives	8
CUFFT	CUDA FFT library	5
FFT	Fast Fourier Transform	1
GPU	Graphics Processing Unit	1
HH	Human hemoglobin	8
PDB	Protein Data Bank	4
SIMD	Single Instruction, Multiple Data	3
VDW	van der Waals	5

References

[1] O. Keskin, A. Gursoy, B. Ma, R. Nussinov, Chemical reviews 108 (2008) 1225. doi:10.1021/cr040409x, [link].
URL http://www.ncbi.nlm.nih.gov/pubmed/18355092

[2] G. Tóth, K. Mukhyala, J. A. A. Wells, Proteins 68 (2) (2007) 551. doi:10.1002/prot.21402.

[3] H. M. Berman, T. N. Bhat, P. E. AU Bourne, Z. Feng, G. Gilliland, H. Weissig, J. Westbrook, Nat Struct Mol Biol (7) (2000) 957. doi:10.1038/80734.

[4] M. Shatsky, O. Dror, D. Schneidman-Duhovny, R. Nussinov, H. J. Wolfson, Nucl. Acids Res. 32 (suppl_2) (2004) W503. doi:10.1093/nar/gkh413.

[5] G. R. Smith, M. J. E. Sternberg, Current Opinion in Structural Biology 12 (1) (2002) 28. doi:10.1016/S0959-440X(02)00285-3.

[6] O. Schueler-Furman, C. Wang, P. Bradley, K. Misura, D. Baker, Science 310 (5748) (2005) 638. doi:10.1126/science.1112160.

[7] J. Janin, Proteins: Structure, Function, and Genetics 52 (1) (2003) 1. doi:10.1002/prot.10398.

[8] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo, I. A. Vakser, Proc Natl Acad Sci USA 89 (6) (1992) 2195. doi:10.1073/pnas.89.6.2195.

[9] R. Chen, Z. Weng, Proteins: Structure, Function, and Genetics 51 (3) (2003) 397. doi:10.1002/prot.10334.

[10] H. A. Gabb, R. M. Jackson, M. J. Sternberg, Journal of Molecular Biology 272 (1) (1997) 106. doi:10.1006/jmbi.1997.1203.

[11] M. J. S. Graham R. Smith, Proteins: Structure, Function, and Genetics 52 (1) (2003) 74. doi:10.1002/prot.10396.

[12] I. A. Vakser, Proteins 29 (S1) (1997) 226. doi:10.1002/(SICI)1097-0134(1997)1+<226::AID-PROT31>3.0.CO;2-O, [link].
URL http://view.ncbi.nlm.nih.gov/pubmed/9485517

[13] A. Tovchigrechko, I. A. Vakser, Nucl. Acids Res. 34 (S2) (2006) W310. doi:10.1093/nar/gkl206.

[14] J. G. Mandell, V. A. Roberts, M. E. Pique, V. Kotlovyi, J. C. Mitchell, E. Nelson, I. Tsigelny, L. F. Ten Eyck, Protein Eng. 14 (2) (2001) 105. doi:10.1093/protein/14.2.105.

[15] R. Grünberg, J. Leckner, M. Nilges, Structure 12 (12) (2004) 2125. doi:10.1016/j.str.2004.09.014.

[16] G. R. Smith, M. J. Sternberg, P. A. Bates, Journal of Molecular Biology 347 (5) (2005) 1077. doi:10.1016/j.jmb.2005.01.058.

[17] M. Frigo, S. G. Johnson, Proc. IEEE 93 (2) (2005) 216. doi:10.1109/JPROC.2004.840301.

[18] NVIDIA GPU computing developer home page, last visit on 22/05/2010.
URL http://developer.nvidia.com/object/gpucomputing.html

[19] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007.