2

Recent Advances and Future Trend on the Emerging Role of GPUs as Platforms for Virtual Screening-Based Drug Discovery

Horacio Pérez-Sánchez, José M. Cecilia and José M. García
Computer Engineering Dept., University of Murcia
Spain

1. Introduction

Virtual Screening (VS) has played an important role in drug discovery, and experimental techniques are increasingly complemented by numerical simulation (Schneider & Böhm, 2002). Although VS methods have been investigated for many years and several compounds could be identified that evolved into drugs, the impact of VS has not yet fulfilled all expectations. Neither the docking methods nor the scoring functions used presently are sufficiently accurate to reliably identify high-affinity ligands. To deal with a large number of potential candidates (many databases comprise hundreds of thousands of ligands), VS methods must be very fast and yet identify "the needles in the haystack". In many VS applications the predicted ligands turn out to have low affinity (false positives), while high affinity ligands rank low in the database (false negatives). In contrast, established simulation (not scoring) methods, such as free-energy perturbation theory, can determine relative changes in the affinity when ligands are changed slightly (group substitutions). However, these techniques require hundreds of CPU hours for each ligand, while simulation strategies to compute absolute binding affinities require thousands of CPU hours for each ligand (Wang et al., 2006). In comparison to these techniques, VS methods must make significant approximations regarding the affinity calculation and the sampling of possible receptor complex conformations. These approximations would be justifiable, as long as the relative order of affinity is preserved at the high-affinity end of the database.

Nowadays there are several receptor based VS methods available, including AutoDock (Zhang et al., 2005), FlexX (Xing et al., 2004), Glide (Friesner & Banks, 2004), FlexScreen (Kokh & Wenzel, 2008), and ICM (Bursulaya et al., 2003), each of them having different technical features. Most modern methods use an atomistic representation of the protein and the ligand. They permit the exploration of thousands of possible binding poses and ligand conformations in the docking process. As a result, binding modes are predicted reliably for many complexes. In general, methods that permit continuous ligand flexibility are somewhat more accurate than those that select conformations from a finite ensemble. However, recent unbiased comparative evaluations of affinity estimations showed little correlation between measured and predicted affinities over a wide range of receptor-ligand complexes. As a result, enrichment rates remain poor for many methods. Since high-accuracy, but also high-cost simulation protocols for affinity calculations are available, one avenue to improve weaknesses

of existing VS programs is to move into the direction of established all-atom biophysical simulation. Both better scoring functions and novel docking strategies will contribute in this direction.

But one of the main problems in this direction is that all VS methods mentioned previously were developed for and run on commodity PCs, with its limitations in terms of availability of computing power. However, current PCs are becoming powerful desktop machines with beyond a teraflop available on them, thanks to the availability of GPUs as an underlying hardware for developing general-purpose applications. We report and show how combining accurate and transferable biophysical VS techniques with these modern massively parallel hardware, allowing significant steps towards more accurate VS screening methods.

Driven by the demand of the game industry, Graphics Processing Units (GPUs) have completed a steady transition from mainframes to workstations to PC cards, where they emerge nowadays like a solid and compelling alternative to traditional computing platforms, delivering extremely high floating point performance and massively parallelism at a very low cost, and thus promoting a new concept of the High Performance Computing (HPC) market; i.e. high performance desktop computing.

This fact has attracted many researchers and encouraged the use of GPUs in a broader range of applications, where developers are required to leverage this technology with new programming models which ease the developer's task of writing programs to run efficiently on GPUs (Garland et al., 2008).

NVIDIA and ATI/AMD, two of the most popular graphics vendors, have released software components which provide simpler access to GPU computing power. CUDA (Compute Unified Device Architecture) (NVIDIA, 2011) is NVIDIA's solution as a simple block-based API for programming; AMD's alternative is called Stream Computing (ATI/AMD, 2011). Those companies have also developed hardware products aimed specifically at the scientific General-Purpose GPU (GPGPU) computing market: Tesla products are from NVIDIA, and Firestream is AMD's product line. More recently (in 2008), the OpenCL[1] framework have emerged as an attempt to unify all of those models with a superset of features, being the first broadly supported multi-platforms data-parallel programming interface for heterogeneous computing, including GPUs and similar devices.

Although these efforts in developing programming models have made great contributions to leverage GPU capabilities, developers have to deal with a massively parallel and throughput-oriented architecture (Garland & Kirk, 2010), which is quite different than traditional computing architectures. Moreover, GPUs are being connected with CPUs through PCI Express bus to build heterogeneous parallel computers, presenting multiple independent memory spaces, a wide spectrum of high speed processing functions, and communication latency between them. These issues drastically increase scaling to a GPU-cluster, bringing additional sources of latency.

Programmability on these platforms is still a challenge, and thus many research efforts have provided abstraction layers avoiding to deal with the hardware particularities of the GPUs and also extracting transparently high level of performance. For example, libraries interfaces for programming with popular programming languages like "Jacket" for Matlab[2],

[1] http://www.khronos.org/opencl/
[2] http://www.accelereyes.com/

or "PyCUDA" or "PyOpenCL" for Python (Klöckner et al., 2011). Some abstraction layers to automatically extract the inherent parallelism existing in many dense linear algebra algorithms, (Agullo et al., 2009), and some of them included as a subroutines in CUDA implementations (Garland et al., 2008; Volkov & Demmel, 2008).

In this review, we give a brief introduction to GPUs in Section 2. Next, we discuss in Section 3 different programming strategies presented in the literature used to overcome the main limitations of VS methods using GPUs, also analyzing their main strengths and weaknesses on single-and-multi GPU-based systems. We also describe in Section 4 our contributions on this field, and finally report our main conclusions about current trends and future predictions in Section5.

2. The roadmap of GPUs as high performance platforms

The graphics hardware has been an active area of research for developing general-purpose computation for many years. The first graphics-oriented machines in which some general purpose applications where developed were the Ikonas (England, 1978), the Pixel Machine (Potmesil & Hoffert, 1989) and Pixel-Planes 5 (Rhoades et al., 1992). These early graphics hardware were typically graphics compute servers rather than desktop workstations.

Moreover, other attempts were made after the wide deployment of GPUs, but still with fixed-function pipelines that were categorized in Trendall & Stewart (2000). For instance, Lengyel et al. (1990) used the rasterization hardware for robot motion planning. Hoff et al. (2001) described the use of z-buffer techniques for the computation of Voronoi diagrams. Kedem & Ishihara (1999) used a graphics hardware to crack the UNIX password encryption. Bohn (1998) also used the graphics hardware in the computation of artificial neural networks. Convolution and wavelet transforms were carried out by Hopf & Ertl (1999), Hopf & Thomas (1999).

However, the milestone to spread GPUs as a general-purpose platform was first motivated by Larsen & McAllister (2001), who demonstrated the capacity of a graphics processor to accelerate a typical dense matrix product through regular texture operators. This result attracted the scientific community into a race for using the GPU as a co-processor, and immediately the number of applications enhanced in that way led to the GPGPU initiative (General-Purpose computation on Graphics Processing Units, also known as GPU Computing and GPGPU.org on the Web) by Mark Harris in 2002 as an attempt to compile all these achievements (Luebke et al., 2006). Soon after, the Cg language was born to ease this path in terms of programmability, trailing earlier achievements like the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex (Lindholm et al., 2001) or fragment processors.

GPUs started to be seriously considered in the HPC community mainly due to the raw performance and massively parallelism of GPUs. The programmable shader hardware was explicitly designed to process multiple data-parallel primitives at the same time. Moreover, GPUs typically had multiple vertex and fragment processors. For instance, the NVIDIA GeForce 6800 Ultra had 6 vertex and 16 fragment processors.

Nevertheless, the graphics hardware was very limited for developing general-purpose applications for several reasons that are mainly summarized in two main constraints: (1) hardware constraints and (2) graphics-devoted programming model. On the hardware side,

the instruction sets of each processor stage were very limited compared to CPU ones; they were primarily math operations, many of which were graphics-specific and only few control flow operations were available. Moreover, each programmable stage could access constant registers across all primitives and also read-write registers per primitive, but these resources were very limited on their numbers of inputs, outputs, constants, registers and instructions. Fragment processors had the ability to fetch data from textures, so they were capable of memory gather. However, the output address of a fragment was always determined before the fragment was processed -the processor cannot change the output location of a pixel-, so fragment processors were initially not able to do memory scatter. Vertex processors evolved acquiring texture capabilities, and thus they were capable of changing the position of input vertices, which ultimately affect where in the image pixels would be drawn. Therefore, vertex processors became capable of both gather and scatter. Unfortunately, vertex scatter could lead to memory and rasterization coherence issues further down the pipeline. Combined with the lower performance of vertex processors, this limited the utility of vertex scatter in the first generation of current GPUs (Owens et al., 2007).

At the beginning of this new GPGPU era in 2002, the available APIs to interact with the GPUs were DirectX 9 and OpenGL 1.4, both of them designed only to match the features required by the graphics applications. To access the computational resources, programmers had to cast their problems into native graphics operations, thus the only way to launch their computation was through OpenGL or DirectX API calls. For instance, to run many simultaneous instances of a compute function, the computation was written as a pixel shader. The collection of input data was stored in texture images and issued to the GPU by submitting triangles. The output was cast as a set of pixels generated from the raster operations with the hardware constraints previously mentioned (Kirk & Hwu, 2010).

Despite of this worst-case scenario, some applications from different scientific fields were ported to the GPU (Owens et al., 2007) by intrepid researchers. Some early work was presented by Thompson et al. (2002) in which they used the programmable vertex processor of an NVIDIA GeForce 3 GPU to solve the 3-Satisfiability problem and to perform matrix multiplication. Besides, Strzodka showed the multiple 8-bit texture channels combination to create virtual 16-bit floating-point operations (Strzodka, 2002), and Harris analyzed the accumulated error in boiling simulation operations caused by the low precision (Harris, 2002) on early generation of GPUs.

Strzodka constructed and analyzed special discrete schemes which, for certain PDE types, allow reproduction of the qualitative behavior of the continuous solution even with very low computational precision, e.g. 8 bits (Strzodka, 2004).

Other efforts were made in fields such as Physically-Based Simulations, Signal and Image Processing, Segmentation, etc (Owens et al., 2007; Pharr & Fernando, 2005).

With the advent of CUDA from NVIDIA in 2006, programming general-purpose applications on GPUs became more accessible. NVDIA has shipped millions of CUDA-enabled GPUs to date. Software developers, scientists and researchers are finding broad-ranging application fields for CUDA, including image and video processing, computational biology and chemistry, fluid dynamics simulation, CT image reconstruction, seismic analysis, ray tracing and many more (CUD, 2011; Hwu, 2011; Nguyen, 2007; Sanders & Kandrot, 2010).

2.1 GPU computing with CUDA

The increasing popularity and promising results of the *GPGPU* within the field of HPC was leveraged by few intrepid programmers for several reasons: (1) the tough learning curve, particularly for non-graphics experts, (2) the high-potential overhead presented by graphics API, (3) the highly constrained memory layout and access model and (4) the bandwidth requirements of multipass rendering.

The advent of CUDA (Compute Unified Device Architecture) from NVIDIA in November 2006, with a new parallel programming model and instruction set architecture, democratized the *GPGPU* (Luebke, 2007), springing up a new era into the community coined *GPU Computing*. *GPU Computing* means using GPUs for computing through parallel programming language and API; i.e. without using the traditional graphics API and pipeline previously introduced.

CUDA leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems without dealing with graphics particularities of the GPU and simply programming in C or C++ with some minimal set of language extensions that are exposed to the programmer. In addition, CUDA comes with a software environment that allows developers to use different high-level programming languages, such as CUDA FORTRAN, OpenCL, DirectCompute[3]. This maintains a low learning curve for programmers familiar with standard programming languages such as C.

CUDA is also a scalable programming model. It is designed to transparently manage tremendous levels of chip-level parallelism through three key abstractions: a hierarchy of thread groups, shared memories and barrier synchronization, providing fine-grained data parallelism and thread parallelism. This scalable programming model has allowed CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions. NVIDIA provides three different GPU products: GeForce, Quadro and Tesla, which are devoted to different markets (NVIDIA, 2011). The latter is the bet of NVIDIA for the HPC market, enhancing the double-precision performance, increasing the memory partitions, enabling error detection, among others to mention but a few.

2.2 CUDA programming model

The increasing popularity of the CUDA programming model (NVIDIA, 2011) is mainly because it presents two main features previously commented: the scalability and the easy-to-use fact. Next we show how these features are developed in the programming model.

A CUDA program is organized into two different subprograms: *host program* and *device program* or *kernels*. The host program consists of one or more sequential threads running on the CPU (host), which are in charge of initializing, monitoring and finalizing the execution of the device program.

The device program consists of one or more parallel kernels that are suitable for execution on the GPU. A **kernel** executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads into a grid of thread blocks (see Figure 1). A **grid** is composed of several blocks which are equally distributed and scheduled among all multiprocessors on the GPU. A **block** is a batch of threads which can cooperate together

[3] http://developer.nvidia.com/directcompute

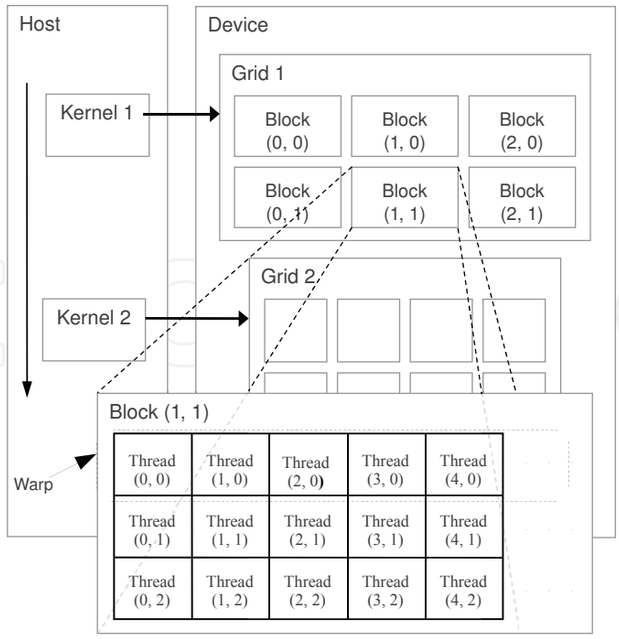


Fig. 1. CUDA programming model

because they are assigned to the same multiprocessor and therefore they share all the resources included in this multiprocessor, such as register file and a high-speed, per-block on-chip memory (called *shared memory*) for inter-thread communication. They are also allowed to synchronize with each other via barriers. Threads from different blocks in the same grid can only coordinate via operations in a shared global memory space (called *device memory*) visible to all threads. The **thread** is the basic execution unit that is mapped to a single core (called *Stream Processor*). Finally, threads included in a block are divided into batches of 32 threads called **warps**. The warp is the scheduled unit, so the threads of the same block are scheduled in a given multiprocessor warp by warp. The programmer declares the number of blocks, the number of threads per block and their distribution to arrange parallelism given the program constraints (i.e., data and control dependencies).

The CUDA programming model presents several patterns of parallelism depending on the previous thread hierarchy. For instance, all threads in a warp execute at the same time the same instruction over different data in a SIMD fashion. However, CUDA requires that thread blocks are independent, meaning that a kernel must execute correctly no matter the order in which blocks are run. Therefore, different thread blocks execute different instructions at a given time, which fit better in a MIMD fashion. This MIMD pattern is one of the key issues in the CUDA programming model as it is the way to ensure scalability, but it also implies that the need for global communication or synchronization amongst threads is the main consideration when it comes to decompose parallel work into separate kernels (Garland et al., 2008).

3. Virtual screening methods on GPUs

In this Section, we summarize the main technical contributions for the parallelization of VS methods on GPUs available on the bibliography. Concretely, we pay special attention to the parallelization of docking methods on GPUs.

In terms of implementations, the trend seems to be reusing available libraries when possible and implement the achievements into existing simulation packages for VS. Among the most-used strategies are either implementing the most time-consuming parts of previously designed codes for serial computers, or redesigning the whole code from scratch. When porting VS methods to GPUs, we should realize that not all methods are equally amenable for optimization. Programmers should check carefully how the code works and whether it is suited for the target architecture. Irrespective of CUDA, most authors maintain that the application will be more accessible in the future thanks to new and promising programming paradigms which are still in the experimental stage or are not yet broadly used. Among them, we may highlight OpenCL or DirectCompute.

3.1 Dock6.2

Yang et al. (2010) present a GPU accelerated amber score in Dock6.2 [4]. They report up to 6.5x speedup factor with respect to 3,000 cycles during MD simulation compared to a dual core CPU. The lack of the single-precision floating point operations in the targeted GPU (NVIDIA GeForce 9800GT) produces small precision losses compared to the CPU, which the authors assume as acceptable. They highlight the thread management utilizing multiple blocks and single transferring of the molecule grids as the main factor that dominates the performance improvements on GPU. They use another optimization techniques, such as dealing with the latency attributed to thread synchronization, divergence hidden and shared memory through tiling, that authors state may double the speedup of the MD simulation. We miss a deeper analysis on the device memory bandwidth utilization. It is not clear whether the pattern accesses to device memory in the different versions of the designs presented here are coalesced or not, which may drastically affect the overall performance.

They finally conclude that the speedup of Amber scoring is limited by the Amdahl's law for two main reasons: (1) the rest of the Amber scoring takes a higher percentage of the run time than the portion parallelized on the GPU, and (2) Partitioning the work among SMs will eventually decrease the individual job size to a point where the overhead of initializing an SP dominates the application execution time. However, we do not see any clear evaluation that supports these conclusions.

3.2 Autodock

Kannan & Ganji (2010) migrate part of a molecular docking application, *Autodock* to NVIDIA GPUs. Concretely, they only focus on the Genetic Algorithm (GA) which is used to find the optimal docking position of a ligand to a protein. They use single-precision floating point operation arguing that, "GA depends on relative goodness among individual energies and single precision may not affect the accuracy of GA path significantly". All the data relative to the GA state is maintained on the GPU memory, avoiding data movement through the PCI Express bus.

The GA algorithms need random numbers for the selection process. They decide to generate the random numbers on the CPU instead of doing on the GPU. The explanation of that is two-fold according to the authors: (1) It enables one to one comparisons of CPU and GPU results, and (2) it reduces the design, coding and validation effort of generating random

[4] http://dock.compbio.ucsf.edu/DOCK_6/

numbers on GPU. From our point of view this decision contradicts the previous assumption of maintaining the data on the GPU, and we do not see enough arguments on these two sentences.

A very nice decision is what the authors call *CGPU Memory Manager* that enables alignment for individual memory request, support for pinned memory and join memory transfer to do all of them in just one transfer. Regarding the fitness function of the GA, authors decide to evaluate all the individuals in a population regardless of modifications. This avoids warp divergences although it makes some redundant work.

Three different parallel design alternatives are discussed in this regard. Two of them only differ in the way they calculate the fitness function, assigning the calculation of the fitness of an individual either to a GPU thread or GPU block. A good comparison between them is provided. The last one includes an extra management of the memory to avoid atomic operations which drastically penalizes the performance.

All of these implementations are rewarded with up to 50x in the fitness calculation but they do not mention anything about global speedup of the Autodock program.

3.3 FFT-based rigid docking

Feng et al. (2010) use a FFT-based method to predict rigid docking configurations, achieving up to 3x speedup factor with its sequential counterpart version. However, FFT is not well suited to be implemented on GPUs, as long as more computations are not being developed afterwards. Moreover, the best implementation of FFT on GPUs (Volkov & Kazian, 2008) up to now is not referenced in this paper. Therefore, it is not clear whether the authors have overcome this implementation or not, and the real benefits of taking this approach for docking.

3.4 Multiple GPU docking

Roh et al. (2009) propose the parallelization of a molecular docking system on GPUs. They obtain 33 to 287 speedups for the calculation of electrostatics and van der Waals energies using different strategies and scaling the number of GPUs until reach two GPUs. Finally, global speedups of up to 2 times are achieved compared to a sequential counterpart version. However, they do not show any practical application of their code. They highlight that an efficient strategy to leverage the power of multiple GPU system is necessary. They also report the importance of an efficient visualization method.

3.5 Genetic algorithms based docking

Korb et al. (2011) enhance the PLANTS (Korb et al., 2006) approach for protein-ligand docking using GPUs. They report speedup factors of up to 50x in their GPU implementation compared to an optimized CPU based implementation for the evaluation of interaction potentials in the context of rigid protein. The GPU implementation was carried out using OpenGL to access the GPU's pipeline and Nvidia's Cg language for implementing the shaders programs (i.e. Cg kernels to compute on the GPU). Using this way of programming GPUs, the programming effort is too high, and also some peculiarities of the GPU architecture may be limited. For instance, the authors say that some of the spatial data structures used in the CPU implementation can not directly be mapped to the GPU programming model because of missing support for shared memory operations (Korb et al., 2011).

The speedup factors observed, especially for small ligands, are limited by several factors. First, only the ligand and protein conformation generation and scoring function evaluation are carried out on the GPU whereas the optimization algorithm is run on the CPU. This algorithmic decomposition implies time-consuming data transfers through PCI Express bus. The optimization algorithm used in PLANTS is the Ant Colony Optimization (ACO) algorithm (Dorigo, 1992). Concretely, authors propose a parallel scheme for this algorithm on a CPU cluster, which use multiple ant colonies in parallel, exchanging information occasionally between them (Manfrin et al., 2006). Developing the ACO algorithm on the GPU as it has been shown in Cecilia et al. (2011) can drastically reduce the communications overhead between CPU and GPU.

3.6 Binding site mapping

Another key point in docking applications is the prediction or estimation of regions on a protein surface that are likely to bind a small molecule with high affinity.

Sukhwani & Herbordt (2010) present a fast, GPU-based implementation of FTMap, a production binding site mapping program. Both the rigid-docking and the energy minimization phases are accelerated, resulting in a 13x overall speedup of the entire application over the current single-core implementation. While an efficient multicore implementation of FTMap may be possible, it is certainly challenging: they estimate it would require an effort greater than what they have spent on the GPU mapping.

The first step assumes the interacting molecules to be rigid and performs exhaustive 3D search to find the best pocket on the protein that can fit the probe. This step is called rigid docking. The top scoring conformations from this step are saved for further evaluation in the second step. The second step models the flexibility in the side chains of the probes by allowing them to move freely and minimizing the energy between the protein-probe complex.

Overall, this work provides a cost-effective, desktop-based alternative to the large clusters currently being used by production mapping servers. Essential to the success of this work is restructuring the original application in several places, e.g., to avoid the use of neighbor lists.

In the future, they plan on extending this work to a multi-GPU implementation and integrating it into a production web server.

4. Testimonials of porting docking algorithm on GPUs

In this Section, we introduce different success stories of porting docking algorithms to GPUs that we have worked on. We also contribute with some novelties in this field; we have worked in this direction and focused on the parallel implementation, incorporation of new improvements in the underlying VS methodology, and exploitation of the docking program FlexScreen which its sequential version is described in Section 4.1. The different strategies we have followed to tame GPUs for FlexScreen can be also used in any VS method.

In Sections 4.2, 4.3 and 4.4, we describe different approaches we have followed for the acceleration of the calculation of non-bonded interactions. In Figure 2 we can see an overview of the main results obtained for the parallelization of the electrostatics kernel using a full coulomb approach (direct summation) or a grid one. In Section 4.5 we show the implementation of a kernel for the fast calculation of SASA (Solvent Accessible Surface Area), widely used in implicit solvation models.

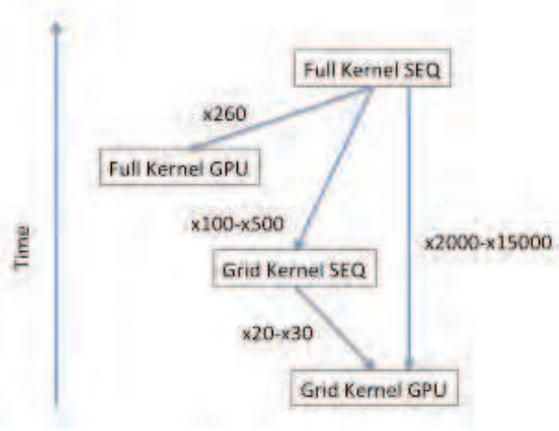


Fig. 2. Comparative of the accelerations obtained using the different kernels in their SEQ (Sequential) and GPU versions for the electrostatics calculation

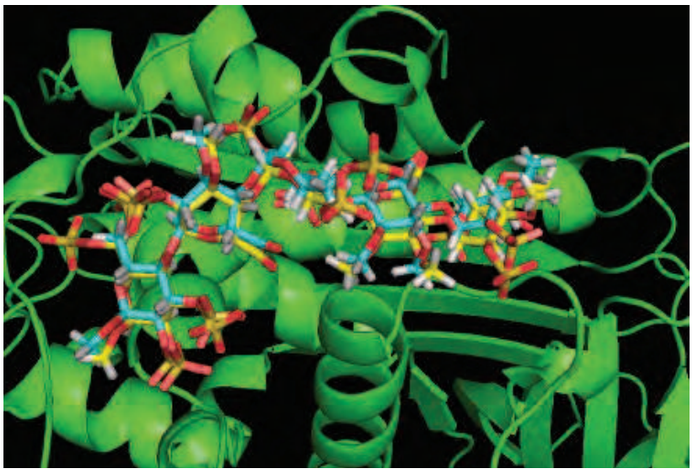


Fig. 3. FlexScreen docking prediction for the binding pose of Heparin with Antithrombin. The model for heparin contains up to 200 atoms and 20 rotatable bonds. Experimental binding pose is yellow colored while FlexScreen prediction is blue colored. Root mean square deviation is less than 1 Å

4.1 Docking with FlexScreen

FlexScreen (Merlitz et al., 2004) performs receptor−ligand docking simulations using an atomistic representation of the protein and the ligand. Ligands are docked using a cascaded version (Fischer et al., 2007) of a stochastic tunneling algorithm (Merlitz et al., 2003) which samples translations of the center-of-mass and rotations of the ligand, as well as intra-molecular conformational changes. In addition to the degrees of freedom of the ligand, receptor conformational change is accounted for in selected side chains. An optimized docked conformation is shown in Figure 3. Previous work demonstrated that this approach yields accurate results for binding mode prediction and improves selectivity in library screens for a number of pharmaceutically relevant receptors (Kokh & Wenzel, 2008).

The FlexScreen scoring function is based on adaptable biophysical models, including electrostatic, Van der Waals, hydrogen bonds and a solvation contribution. For the calculation

of electrostatic and Van der Waals terms during the docking simulation, precomputed grids that represent the protein are used (Meng et al., 1992). FlexScreen is divided into two programs; *dogrid*, which performs the electrostatic (ES) and Van der Waals (VDW) precalculation in form of a grid for a given receptor structure and *dock*, which reads the previously generated ES and VDW grid files and carries out the docking simulation for a single ligand or ligand database.

4.2 Full calculation of the non-bonded interactions

In this section, we focus on the optimization of the full version (direct summation) of the calculation of non-bonded interactions (such as electrostatics and van der Waals forces), as this kernel is an important bottleneck to different VS methods (Perez Sanchez & Wenzel, 2011). On GPUs, Stone et al. (Stone et al., 2007) reached speedups of around 100 times, while Harvey et al. (Harvey & De Fabritiis, 2009) achieve a 200 times acceleration. We test our kernel in GPUs to exploits the parallelism of this application, getting up to 260 times speedup compared to its sequential version.

Algorithm 1 Sequential pseudocode for the calculation of electrostatic interactions for a receptor ligand case, full kernel version (direct summation)

```
1: for i = 0 to nrec do
2:     for j = 0 to nlig do
3:         calculus(rec[i], lig[j])
4:     end for
5: end for
```

In order to exploit all the resources available on the GPU, and get the maximum benefit from CUDA, we focus first on finding ways to parallelise the sequential version of the electrostatic interaction kernel, which is shown in Algorithm 1, where *rec* is the biggest molecule, lig the smallest molecule, *nrec* the number of atoms of *rec* and $nlig$ the number of atoms of lig.

In our approach, CUDA threads are in charge of calculating the interaction between atoms. However, the task developed by the CUDA thread blocks in this application can drastically affect the overall performance. To avoid communication overheads, each thread block should contain all the information related to the ligand or protein. Two alternatives come along to get this. The former is to identify each thread block with information about the biggest molecule; i.e. CUDA threads are overloaded, and there are few thread blocks running in parallel. The latter is exactly the opposite, to identify each thread as one atom of that molecule and then CUDA threads are light-weight, and there are many thread blocks ready for execution. The second alternative fits better in the GPU architecture idiosyncrasy.

Figure 4 shows this design. Each atom from the biggest molecule is represented by a single thread. Then, every CUDA thread goes through all the atoms of the smallest molecule.

Algorithm 2 outlines the GPU pseudocode we have implemented. Notice that, before and after the kernel call, it is needed to move the data between the CPU RAM and the GPU memory.

The kernel implementation is straightforward from Figure 4. Each thread simply performs the electrostatic interaction calculations with its corresponding atom of the *rec* molecule and all the lig molecule atoms.

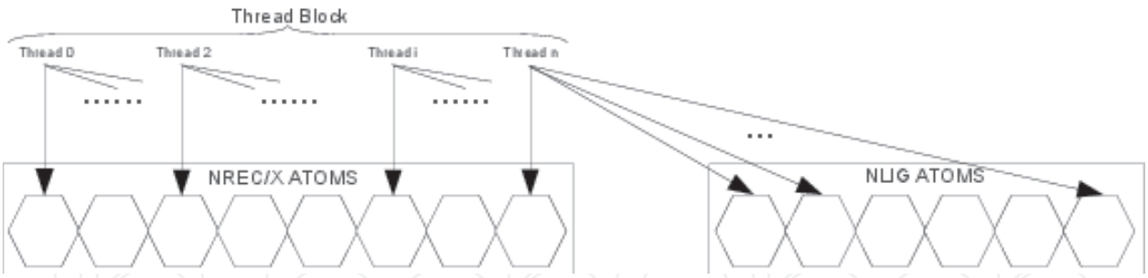


Fig. 4. GPU design for X thread blocks (with $X = 1$) with n threads layout.

Algorithm 2 GPU pseudocode for the full ES kernel.

1: $CopyDataFromCPUtoGPU(rec)$
2: $CopyDataFromCPUtoGPU(lig)$
3: $numBlocks := nrec/numThreads$
4: **Kernel**$(numBlocks, numThreads)$
5: $CopyDataFromGPUtoCPU(result)$

CUDA Kernels

Algorithm 3. Basic implementation

1: **for all** Blocks **do**
2: **for** $i = 0$ to $nlig$ **do**
3: $calculus(myAtomRec, lig[i])$
4: **end for**
5: **end for**

Algorithm 4. Tiles implementation

1: **for all** Blocks **do**
2: $numIt = nlig/numThreads$
3: **for** $i = 0$ to $numIt$ **do**
4: $copyBlockDataToSharedMemory(lig)$
5: $calculusBlock(myAtomRec, ligBlock)$
6: **end for**
7: **end for**

We have derived two different implementations: the basic one (Algorithm 3), and the advanced one (Algorithm 4), where a blocking (or tiling) technique is applied to increase the performance of the application, grouping atoms of the *lig* molecule in blocks and taking them to the *shared memory*, taking advantage in this way of the very low access latency to the *shared memory*.

4.2.1 Performance evaluation

The performance of our sequential and GPU implementations are evaluated in a quad-core Intel Xeon E5530 (Nehalem with 8 MB L2 cache), which acts as a host machine for our NVIDIA Tesla C1060 GPU. We compare it with a Cell implementation (Pérez-Sánchez, 2009) in a IBM BladeCenter QS21 with 16 SPE.

Figure 5 shows the execution times for all our implementations (both GPU and Cell) taking into account the data transfer between the RAM memory and the corresponding device memory. All the calculations are performed in simple precision floating point, due the smaller number of double precision units of the Tesla C1060. The benchmarks are executed by varying the number of atoms of the smallest molecule and also the number of atoms of the biggest molecule for studying the cases of protein-protein and ligand-protein interactions. In Figure

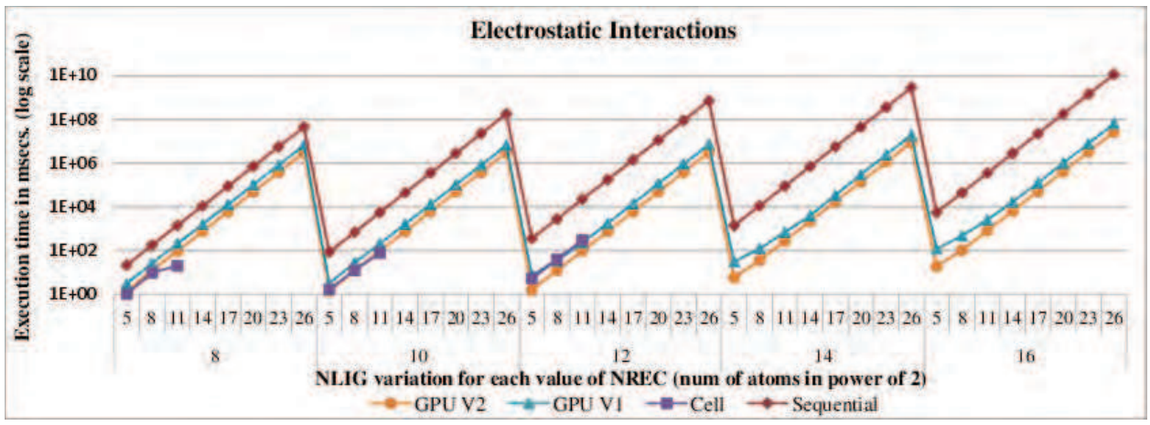


Fig. 5. Results obtained for different molecular size ratios. The execution time for the calculation of the electrostatic potential, in single precision, executed 128 times in a loop for different conformations of the molecule.

5 the performance of the Cell implementation, GPU basic implementation (GPU V1) and GPU tiles implementation (GPU V2) enhances along with the value of *nrec*, defeating the sequential code by a wide margin (up to a speed factor of 260x). Notice that, the speedup factor between GPU and CPU increases faster when the value of *nrec* is higher. It is because the number of thread blocks running in parallel is also higher, and then the GPU resources are fully used. Similarly, for larger values of *nlig*, the speedup factor between GPU and CPU increases also because there are more threads running at the same time. However, it remains flat for a configuration greater than 256 threads per block.

Cell processor is not able to execute some of the biggest benchmarks due to its hardware constraints, mainly related to the 256K SPE Local Storage. However, it performs similarly well compared to the GPUs for the smallest benchmarks in which the GPU is not fully used.

The results obtained for GPU are indeed promising, given the obtained speedup values up to 260x, compared to its sequential version. Cell processor gives similar results to GPUs only in some cases, where the molecules are small and the saturation situation for the GPU is not reached, but for higher workloads GPUs attain speedup values 7 times higher than the Cell processor. This way we can work with bigger molecules and thus perform more realistic calculations.

4.3 Precomputation of grids

In the recent years, the completion of the human genome project has brought new and still unprocessed information about potential targets for the treatment of human diseases with drugs. It is well known that the efficacy of a drug can be vastly improved through the interaction with multiple targets, although undesirable side effects must also be studied. Therefore, it is very important to identify and validate all potential targets for a given compound. Experimental approaches for this purpose are very expensive and time consuming, while in-silico approaches like Virtual Screening (VS) can efficiently propose accurate predictions that drastically reduce testing procedures in the laboratory.

Multiple target drug screening is a particular case of VS methods. In the approach that we propose, the main bottleneck of the calculations is related with the computation of

non-bonded kernels in a specific way, concretely the precomputation of potential grids. We show in this Section its acceleration by means of GPUs.

From the other side, VS approaches for multiple target identification have not been yet fully explored. Docking methods have been traditionally applied to ligand database screening (Yuriev et al., 2011), where a large ligand database is screened against a single receptor molecule in order to predict potential leads. The inverse approach, the one we are interested in, where a large database of target receptors is screened against a single ligand, has not received such attention and only some attempts are reported (Hui-fang et al., 2010). In both application scenarios, most docking programs represent the receptor as a rigid molecule (Yuriev et al., 2011) thus limiting the range of applicability of their predictions. There are few reported cases where receptor flexibility has been successfully used in docking simulations (Kokh et al., 2011), but it is clear its relevance and importance for multi target drug screening.

For a realistic simulation of one receptor-ligand pair in *FlexScreen* (Section 4.1) it takes around 80% of the total running time for *dogrid* and 20% for *dock*. In the application case of multiple target drug screening we need to screen one ligand against a large database of receptors, therefore the main bottleneck is the generation of grids by *dogrid*. Even more, ES and VDW grids generated by *dogrid* can also be used for protein surface screening or blind docking (Hetényi & van der Spoel, 2002), an approach where no assumption is done about the part of the receptor where the docking starts. In this situation, we need to determine it first, and the fast examination of ES and VDW grids yields valuable information about potential binding sites, as it has already been shown for the discovery of inhibitors for antithrombin (Meliciani et al., 2009; Navarro-Fernandez et al., 2010). Therefore, our main interest in order to achieve an optimized implementation of multiple target drug screening with flexible receptors is to target our efforts to the acceleration of *dogrid*.

4.3.1 Code design

In this part, we introduce several different GPU designs for the generation of the electrostatic (ES) and Van der (VDW) Waals grids. Firstly, the CPU baselines of that generation are presented before introducing the GPU design proposal. Calculations are always carried out in double precision floating point.

4.3.1.1 Grid calculation on the CPU

In the sequential version of FlexScreen, precomputation of electrostatic (ES) and Van der (VDW) Waals grids is performed in the *dogrid* program as follows; the protein is placed inside a cube of minimal volume $Vol = L^3$ that encloses the protein. A three dimensional grid is created dividing the cube into $(N-1)^3$ smaller cubes of identical volume, each one of side length $d = L/N$, so that the total number of grid points is N^3.

$$V_{elec,i} = \sum_{j=1}^{NREC} \frac{q_j}{r_{ij}} \tag{1}$$

The electrostatic potential due to all protein atoms is calculated on each grid point i according to the Coulomb expression given by equation 1. The total number of atoms of the protein is equal to $NREC$, while q_i is the charge of each individual atom i of the receptor and r_{ij} the distance between point i of the grid and atom j of the receptor. This information is represented

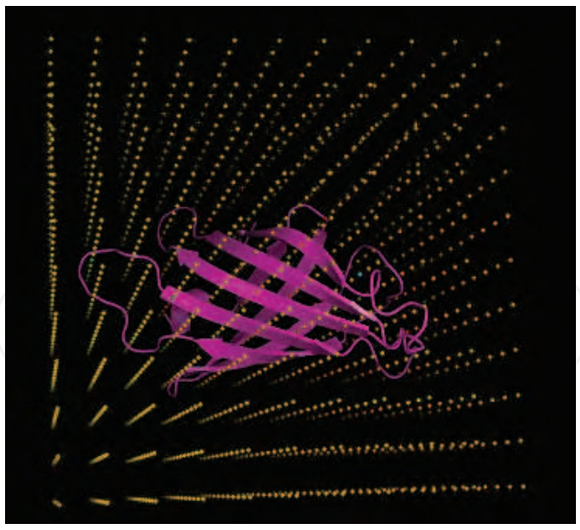


Fig. 6. Grid for streptavidin. Length of the side of the cube (L) is 50 Å, spacing between grid points (d is 5 Å, and the total number of grid points is equal to 11^3

Algorithm 5 Sequential pseudocode for ES grid

```
1: for i_x = 1 to N do
2:     for i_y = 1 to N do
3:         for i_z = 1 to N do
4:             for j = 1 to NREC do
5:                 calculus(rec[j], ESgrid[i_x, i_y, i_z])
6:             end for
7:         end for
8:     end for
9: end for
```

by $rec[j]$ for all receptor atoms. The pseudocode is shown in Algorithm 5, where *calculus* performs the calculation of the electrostatic potential following equation 1 for each grid point (defined by its grid indexes i_x, i_y and i_z) computing the non-bonded interactions against all protein atoms and storing conveniently the values in the ES grid file.

$$V_{vdw,i} = 4\epsilon_{ij} \sum_{j=1}^{NREC} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \quad (2)$$

The calculation of the VDW potential in the *dock* program is performed following equation 2 where ϵ_{ij} and σ_{ij} are the VDW OPLS force field parameters (Jorgensen, 1996) that depend on the type of interacting atoms. Given the fact that the VDW potential decays very fast at short distances, it is convenient to define a cutoff radius r_{cutoff}. Then, we calculate in the scoring funcion the VDW potential only in the cases where atoms are closer than this distance, since for larger values it is very close to zero. Avoiding calculation in those cases we can speed up the global VDW computation. The default r_{cutoff} value used is 4 Å. For this purpose the VDW grid is precalulated in *dogrid*, and it contains on each VDW grid point only information about the indexes of all protein atoms that fulfill this distance condition around each grid

Algorithm 6 Sequential pseudocode for VDW grid

```
1: for i_x = 1 to N do
2:   for i_y = 1 to N do
3:     for i_z = 1 to N do
4:       for j = 1 to NREC do
5:         if isNeighbour(i_x, i_y, i_z, rec[j]) then
6:           store(VDWgrid[i_x, i_y, i_z], j)
7:         end if
8:       end for
9:       sort(VDWgrid[i_x, i_y, i_z], MAXNB)
10:    end for
11:  end for
12: end for
```

point. As shown in Algorithm 6, once r_{cutoff} is defined, for each grid point (i_x, i_y, i_z) we check individually against all the NREC receptor atoms which of them are closer than this distance. Once this check is finished, we sort them and store only the closest $MAXNB$ neighbouring atoms. We have previously tested that a value of 20 for $MAXNB$ yields accurate results for VDW energies.

4.3.1.2 Grid calculation on GPU

We describe in this part the strategy we followed for the calculation of the ES and VDW grids on the GPU.

Algorithm 7 GPU pseudocode for Electrostatic (ES) grid

Host (CPU)

```
1: CopyDataCPUtoGPU(rec)
2: nBlocks := ngp/numThreads * NREC/AT_BLOCK
3: Kernel <<< nBlocks, numThreads >>> (rec, ESGrid)
4: CopyDataFromGPUtoCPU(ESGrid)
```

Device (GPU)

```
1: for all nBlocks do
2:   copyBlockDataToSM(rec, AT_BLOCK)
3:   for i = 0 to AT_BLOCK do
4:     calculus(energy, rec[i])
5:   end for
6:   atomicAdd(&ESGrid[myCell], energy)
7: end for
```

Algorithm 7 describes the calculation of the Electrostatic (ES) grid on GPU. Firstly, all information related to the receptor (atomic positions and partial charges) and represented by *rec* in the pseudocode is copied from CPU's host memory to GPU's device memory. Moreover, the ES grid is allocated on device memory.

The ES grid is divided into thread-blocks, having as many thread-blocks as the total number of grid points ($ngp = N^3$) divided by the number of threads ($numThreads$), which is actually a degree of freedom. Moreover, per each of these thread-blocks, we set $NREC/AT\_BLOCK$ thread-blocks, being AT_BLOCK (number of atoms per block) a fixed value which is a degree of freedom as well.

All threads in a thread-block collaborate to obtain a coalesced access to device memory, and also to prepare a tiling technique. The former is guaranteed as threads in the same warp access to the same memory segment. Moreover, the tiling technique is implemented; .initially

all threads collaborate to store in shared memory (SM) the information pertaining to its block of protein atoms, i.e. AT_BLOCK and afterwards the *calculus* function is performed for a given point of the ES grid, represented by i_x, i_y, i_z. Finally the result is stored back in the ES grid.

It should be noticed that the same data-block from *rec* is loaded as many times as thread-blocks represent the ES grid; i.e. $ngp/numThreads$, but this is done by different thread-blocks. Finally, atomic operations are performed to sum up individual values calculated by the different threads. ES grid data is copied back to host memory and written to disk.

Algorithm 8 GPU pseudocode for Van der Waals (VDW) grid

Host (CPU)

1: $CopyDataFromCPUtoGPU(rec)$
2: $nBlocks := ngp$
3: $KernelVDW <<< nBlocks, numThreads >>> (rec, VDWGrid)$
4: $CopyDataFromGPUtoCPU(VDWGrid)$

Device (GPU)

1: **for all** ngp **do**
2: **for** $i = 0$ to $1NREC/numThreads$
3: **do** $StoreInSM(neighbourhoodShared, i, isNeighbour(i, rec[i], r_{cutoff}))$
4: **end for**
5: **if** $tid == 0$**and** $numNeighbours > MAXNB$ **then**
6: $sort(neighbourhoodShared)$
7: **end if**
8: $VDWGrid[i_x, i_y, i_z] = neighbourhoodShared$
9: **end for**

The calculation of the Van der Waals (VDW) grid on the GPU is described by Algorithm 8. Each thread-block calculates neighbours for each grid point of the VDW grid. Moreover, the *rec* information (atoms of the receptor) is equally divided into threads of each thread-block, assigning different sets of *rec* to each thread. Thus each thread calculates the distance between the grid point (represented by i_x, i_y, i_z) of its thread-block and all atoms in the *rec* set associated with it.

For those atoms closer than cutoff radius r_{cutoff}, threads store their indexes and distance values in an array represented by *neighbourhoodShared*. This process is performed in shared memory to avoid costly accesses to device memory. If the number of neighbours found for a given cell is bigger than the maximum number of neighbour ($MAXNB$), they are sorted and only the $MAXNB$ closest-neighbours are stored as final result for the VDW grid. Finally VDW grid data is copied back to host memory and written to disk.

4.3.2 Performance evaluation

In what follows our hardware platforms are: (1) a dual-socket 2.40 GHz quad-core Intel Xeon E5620 Westmere(R) processor, and (2) a NVIDIA Geforce GTX 465 based on Fermi architecture released in November 2010 (NVIDIA, 2009). We use GNU *gcc* version 4.3.4 with the -O3 flag to compile our CPU implementations, and CUDA compilation tools (release 4.0) on the GPU side.

The code of *dogrid* was profiled using the GNU tool *gprof* (Graham et al., 2004) and by manual introduction of timers in the code, yielding similar results in both cases for different protein sizes and grid densities (represented by N^3). It is desirable to use as many grid points

ngp	Ex. time (CPU)	GFLOPS (CPU)	% Ex. time ES (CPU)	% Ex. time ES (GPU)	Speedup ES	Speedup VDW	Global Speedup	GFLOPS ES	GFLOPS VDW	TOTAL GFLOPS
17^3	171.637	0.764	90.7	33.4	131.26	6.74	48.42	62.64	30.94	37.2
33^3	1372.59	0.77	90.67	27.24	155.09	5.97	46.6	74.04	27.22	37.7
65^3	10908.779	0.77	91.2	31.97	160.2	7.27	56.17	76.5	35.3	46.2
129^3	86950.656	0.78	91.7	35.8	163.13	8.27	63.7	77.1	42.26	52.96
193^3	290558.25	0.77	91.7	36.9	159.5	8.45	64.2	76.2	43.7	54.34

Table 1. Results obtained for different grid densities (specified by ngp, number of grid points) for the protein streptavidin (1740 atoms) in a cubic grid of volume 32^3 Å^3. Ex. time means execution time in milliseconds.

as possible for the grid, since interpolation strategies are used later in the *dock* program to calculate the Van der Waals and electrostatic energies in the scoring function. Higher number of grid points imply smaller spacing between grid points and therefore more accuracy for the interpolation procedure. However, the size of the grid grows with N^3 and consequently the necessary memory storage also increases. Nevertheless, we have found that a satisfactory approach consists in the use of already tested average grid spacing values that yield good accuracy in the docking calculations. We have tested it and found that a grid spacing value of 0.5 Å gives a good compromise between accuracy and memory requirements. In these cases, ES grid calculation takes around 80 % of the *dogrid* running time while the calculation of the VDW grid takes around 20 %. Less than 1 % of the time is involved in input file reads and final grid file writes. According to Amdahl's law (Amdahl, 1967) it is clear that if we focus on individual acceleration of both ES and VDW grid calculations and succeed, global *dogrid* program would achieve high speedups.

We summarize in this part the main results obtained in our GPU implementation.

Table 1 shows different performance parameters obtained with our CPU (*dogrid* program) and GPU versions of the ES and VDW grid calculations for the protein streptavidin. In the different columns, we specify the total number of grid points used in the grid, the percentage of time spent in the ES grid computations (for VDW grid it can be inferred substracting it from 100 %) in both CPU and GPU versions, the speed-up factor obtained by the GPU grid calculations of ES and VDW grids compared to the sequential counterpart version, and finally the maximum values of GFLOPS obtained by our GPU codes. It is noteworthy to remark that we count the *sqrt* and *mad* operations as a single and double FLOP respectively. When the number of grid points increases, performance of the sequential version remains constant, while the performance of the GPU implementation slightly increases reaching saturation values.

The maximum speed-up factor attained by the ES grid calculation for streptavidin reaches 163x, while for the VDW grid calculation we report a maximal 8x speed-up factor. The lower speedup value obtained for the latter kernel is due to the less arithmetic-intensity kernel and to a higher number of synchronization constraints than in the former. Global speedups for *dogrid* attain accelerations in the $50 - 60$ speedup range. It is also worth mentioning that the calculations have been performed in double precision. We have checked that switching from double to single precision in the Fermi GPU changes ES grid speedup factor from 160 to 250 times. Since same memory can be now filled with two times more protein information, less time is involved in data transfer and more in computations in single floating point arithmetic, which is faster than for double precision, both factors contributing to the higher speedup. Nevertheless, we decided to work always in double precision for the grid generation, given the required accuracy for the docking simulations.

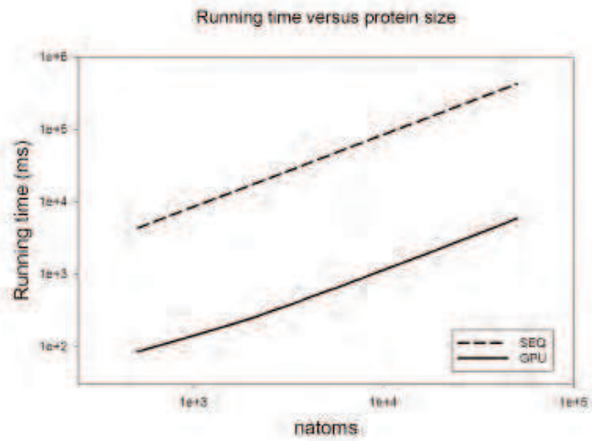


Fig. 7. Running times in miliseconds for the ES and VDW grid calculations obtained with the sequential version (dashed line) and GeForce GTX 465 GPU version (continous line) versus the total number of atoms (natoms) taken from the protein. Both axes are in logarithmic scale. Protein chosen is mammalian fatty acid synthase (PDB ID: 2UVZ)

In order to measure the performance of our parallel implementation we calculated the number of floating point operation per seconds, specified in terms of GFLOPS, for double precision. For the best cases we have obtained a maximum of 77 and 43 GFLOPS for both ES and VDW grid implementation, which is translated into a global 53 GFLOPS performance measure for the whole program, clearly outperforming the performance of 0.7 GFLOPS obtained by the sequential version. Nevertheless, we think that there is still room for improvement in our implementation, concretely for the VDW grid GPU kernel.

Previous results from Table 1 have been obtained for streptavidin, medium-small size protein, but we have checked that for bigger proteins and the global acceleration results remain in the same range. Regarding applicability range of our implementation, the usual protein sizes involved in drug screening tend to be between 1000 to 100000 atoms. We have studied how does our implementation behave in this range of receptor sizes. In Figure 7, we mesaure total running time for the generation of ES and VDW grids for both the sequential and GPU (GeForce GTX 465) implementations. We have chosen mammalian fatty acid synthase as study protein since with 60000 atoms it is one of the biggest proteins feasible for docking calculations. We have performed our calculations varying the number of atoms used in the grid computations. It can be clearly seen that a two orders of magnitude speedup is obtained for the GPU implementation over the whole protein size range, so we are sure that our implementation is valid in a long protein size range as happen usually in multi target drug screening calculations.

In this Section, we have efficiently shown how the CUDA language can be used to exploit the GPU architecture in an applied drug discovery problem. At this point and as far as we know, this is the first GPU implementation of a multiple target drug screening methodology.

We have accelerated the grid generation of the docking program FlexScren for multiple target drug screening using the CUDA language for the GPU architecture. We have obtained average speedups of up to 160 and 8 times for the acceleration of ES and VDW grid calculations for a

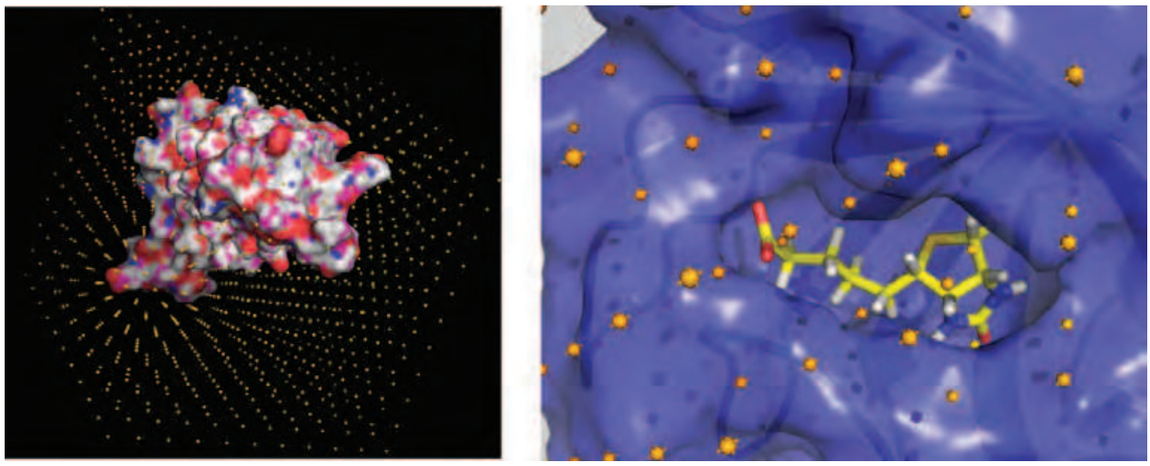


Fig. 8. (A) Representation of the grid for the protein streptavidin. Length of the side of the cube (L) is 50 Å, spacing between grid points d is 5 Å, and the total number of grid points is equal to 11^3. (B) Biotin in the binding pocket of streptavidin

range of proteins in the $1000 - 10000$ atoms size range with high accuracy in double floating point precision. These are translated to global speedups of up to 60 times for the program *dogrid*.

4.4 Calculation of non-bonded interactions using grids

We have described in Section 4.2 how the bottleneck of VS methods are related with the computation of full non-bonded interactions Kernels and how GPUs can yield speedups of up to 260 times (Guerrero et al., 2011). Nevertheless, mentioned Kernels need to perform N^2 interactions calculations (N = total number of particles in the system) and even using GPUs, the required computation time grows polynomically with N so this imposes serious limitations for the simulation of big size systems. Thus we decided to look for alternatives to full Kernels and decided to use grid Kernels (Meng et al., 1992). We have checked that just in the sequential version, speedups of 200 times versus the full non-bonded Kernel are obtained. In Section 4.3 we have reported how the calculation of the grids is performed.

We describe in this Section how to unleash the potential of GPUs for the calculation of non-bonded potentials in VS using grids. Previous works have investigated this approach in a similar fashion but for long range interactions using Ewald-Mesh methods (Cerutti et al., 2009). Given the molecular sizes involved in protein-ligand interactions, we are only interested in short-range electrostatics. Related works reported a 3 times speedup using a different approach (Feng et al., 2010), a 50 times speedup focusing on the acceleration of more particular Kernels of the docking program Autodock (Kannan & Ganji, 2010), and a 7 times acceleration of the Dock6 scoring function by Yang et al. (2010).

The protein is placed inside a cube of minimal volume $Vol = L^3$ that encloses it. A three dimensional grid is created dividing the cube into $(N-1)^3$ smaller cubes of identical volume, each one of side length $d = L/N$, so that the total number of grid points is N^3. The electrostatic potential due to all protein atoms is calculated on each grid point i according to the Coulomb expression (Meng et al., 1992). A graphical depiction of the grid for streptavidin can be seen in Figure 8(A), and in more detail for the ligand biotin on its binding pocket in Figure 8(B).

Once the protein grid is loaded into memory, the calculation of the electrostatic potential for the protein-ligand system is performed as follows: for each ligand atom i with charge q_i at point P_i we calculate which are the eight closest protein grid point neighbours. Next, an interpolation procedure is applied to estimate the value of the electrostatic potential due to all protein atoms at P_i. The same procedure is applied to all ligand atoms summing them up. Different interpolation procedures in 3D have been used (Press et al., 1992); linear, cubic and Gauss interpolation.

4.4.1 Code design

In this Section, we introduce the CPU and GPU designs for the calculation of the electrostatic (ES) potential using grids. We have used NVIDIA's CUDA (NVIDIA, 2010) for the GPU implementations on two different machines; a) a host Intel Xeon E6850 CPU with a NVIDIA GeForce GTX 465 GPU and b) a host Intel Xeon E5620 with a NVIDIA Tesla C2050 GPU. They are referred to as Fermi and Tesla. We use GNU *gcc* version 4.3.4 with the -O3 flag to compile our CPU implementations, and CUDA compilation tools (release 4.0) on the GPU side.

4.4.1.1 ES energy calculation on CPU

Algorithm 9 Sequential pseudocode for the calculation of the electrostatic potential

```
1: for i = 1 to N do
2:     for j = 1 to nlig do
3:         energy[i * nlig + j] =
           interpolate(lig[i * nlig + j], ESGrid)
4:     end for
5: end for
```

We perform a VS experiment where a ligand database containing up to thousands of ligands is screened against a single protein molecule. The precomputed protein grid is read from file and loaded onto memory. Next, the electrostatic (ES) energy of each atom is calculated using interpolation on the grid as explained before and following the pseudocode shown in Algorithm 9, where N is the number of ligands, $nlig$ is the number of atoms of each ligand and the function $interpolate$ performs the calculation of the electrostatic potential for each atom.

4.4.1.2 ES energy calculation on GPU

We describe in this part the different strategies studied for the GPU implementation. All designs have in common that one thread calculates the energy of only one atom. The threads are organized in blocks of fixed size $numThreads$, being this an important optimization parameter.

1. **GM** (use of device memory): In this initial design the whole grid is stored in the GPU device memory. No additional optimizations are implemented.
2. **ROI** (truncation to the grid around the ligand): In this strategy we have implemented some optimizations with respect to the previous version (GM). In order to reduce the CPU-GPU data transfer time, we can take advantage of the fact that we only need to access the grid positions in the volume where the ligand is enclosed. Thus, we can define a region of interest (ROI) of the grid around the ligand and send only that part to the GPU instead of the whole grid.

3. **ZIP** (compression of the grid): In addition, we can shorten the CPU-GPU grid transfer time with the compression of the positions of the ligand atoms. For that purpose, we discretize the volume that encloses the ligands in another cubic regular grid, where each ligand atom is specified only by its grid cell index. The advantage of this approach is that if we perform a fine grain division, each atom can be conveniently represented by just 3 short integers (instead of the three doubles required for position) and reduce memory usage to a quarter.
4. **SM** and **TM** (use of shared and texture memories): We can benefit from the use of shared and texture memory to improve memory access. In the shared memory approach (SM), threads of a block cooperate to copy ROI of the grid to the shared memory in order to obtain a lower memory access penalization. It must be noticed that accessing grid data on SM is coalesced in order to leverage memory bandwith. Regarding texture memory (TM) approach, we can store protein grid into the texture memory unit (TMI), so that just accessing different memory indexes gives us directly the interpolated (NVIDIA, 2010) energy value for each atom.

Algorithm 10 GPU pseudocode for the calculation of the electrostatic potential

Host (CPU)

1: $CopyDataCPUtoGPU(GridROI)$
2: $clig = compress(lig)$
3: $CopyDataCPUtoGPU(clig)$
4: $nBlocks := N * nlig/numThreads$
5: $Kernel <<< nBlocks, numThreads >>> (GridROI, clig, energy)$
6: $CopyDataFromGPUtoCPU(energy)$

Kernel 1: ROI-TM-ZIP

1: **for all** nBlocks **do**
2: $dlig = decompress(clig[myAtom])$
3: $ilig = positionToROICoordinates(ROIinfo, dlig)$
4: $energy[myAtom] = accessToTextureMemory(GridROI, ilig)$
5: **end for**

Kernel 2: ROI-SM-ZIP

1: **for all** nBlocks **do**
2: $copyDataToSM(GridROI)$
3: $dlig = decompress(clig[myAtom])$
4: $ilig = positionToROICoordinates(GridROI, dlig)$
5: $energy[myAtom] = interpolate(GridROI, dlig, ilig)$
6: **end for**

Kernels shown in Algorithm 10 describe two different mixed groups of optimizations based on the previous strategies. In both Kernels, the host sends the ROI of the grid and the compressed ligand atom positions to the GPU. In Kernel 1 of Algorithm 10, each thread decompresses the coordinates of the corresponding atom and calculates its coordinates in the ROI coordinates system. Next, it performs interpolation in the TMI. In Kernel 2 of Algorithm 10, each thread also decompresses coordinates but the interpolation function is implemented in the code as described. Finally, energy values are copied back to CPU.

4.4.2 Grid spacing and interpolation accuracy

Figure 9 shows how grid spacing d influences accuracy in the different interpolation procedures. We wondered about the smallest possible value of d that yields good accuracy

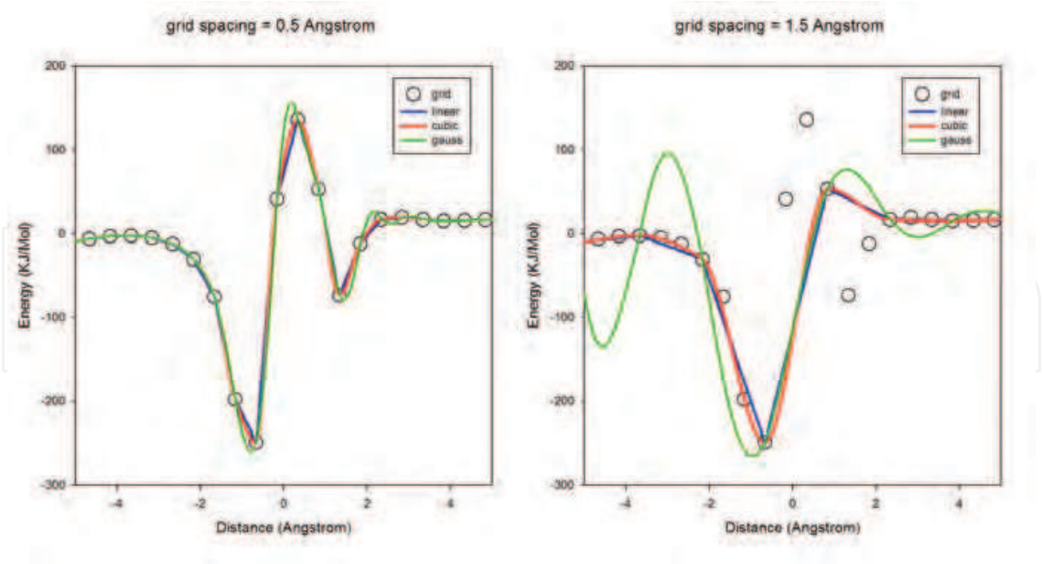


Fig. 9. Interpolation results obtained in a part of the grid streptavidin-biotin for grid spacing values of 0.5 (left picture) and 1.5 (right picture) Å, and using different interpolation procedures. For clarity of the comparison we show the values for the grid points pertaining to a grid with a spacing of 0.5 Å.

and that uses the less possible number of points for the grid, and therefore memory. We found that a value of $d = 0.5$ Å gives good accuracy for the three interpolation methods. Smaller values of d do not improve accuracy significantly while they require more memory (it depends on $(1/d)^3$). From the other side, higher values of d, like 1.5 Å yield unacceptable results for all studied interpolation methods in the rugged parts of the curve, which happens often due to the typical charge distribution in proteins. Therefore we accepted a value of $d = 0.5$ Å as optimal. Regarding the interpolation procedure we discarded the Gauss interpolation given its worst results in the rugged parts of the curve, if we compare it with cubic and linear interpolations, which yield similar accuracy. We finally decided to use only the latter given its lower computational cost. We also discarded interpolation methods of higher order since in the ROI strategy (grid is reduced around the ligand) they would not be able to access grid points out of ROI, yielding wrong results.

4.4.3 Analysis and performance of the sequential code

In Figure 10(B), we can see the timing results obtained for the sequential code in a ligand database screening with 2000 ligands. The trilinear interpolation needs to access eight adjacent cells of a ligand atom positions. It implies two memory accesses to four different rows of the grid. Furthermore, we cannot exploit the use of the cache due to the fact that the atoms are spread in random positions in the 3D space. Therefore most of the RAM accesses represent a bottleneck. Nevertheless, we have used this grid Kernel as starting point and investigated how to adapt it to the GPU architecture, since it is widely used in most biomolecular simulation methods. An additional reason is the 150 to 200 speedups in the sequential version versus the full kernel (Guerrero et al., 2011) for several grid densities and size ranges of rigid proteins. Besides, the GPU computational time is divided in Figure10(A) between time

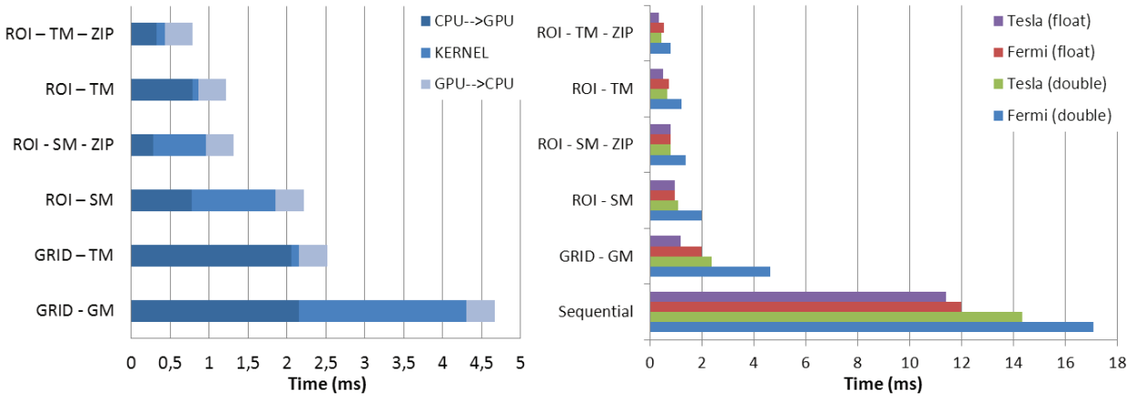


Fig. 10. (A) Comparison of running times for the sequential and GPU implementations. Protein studied is streptavidin and the screening was performed using a ligand database containing 2000 ligands, each one containing around 32 atoms. (B) Total running times for the two GPUs used in our study in float and double precision.

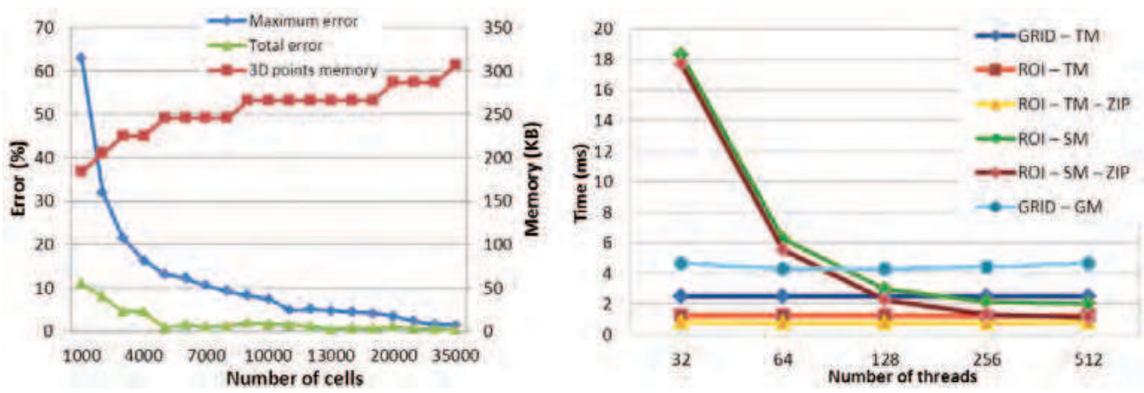


Fig. 11. (A) Values of the maximum and total error per atom obtained when using the compressed grid for representing the ligand database for several values of the number of cells and memory consumption in KB (B) Influence of the number of threads per block on the running time for the different implementation strategies studied.

dedicated to computation and memory transfers between CPU and GPU through PCI Express bus.

4.4.4 Compression of the ligand database atomic positions

In Figure 11(A) we can observe how the number of cells influences the error for the calculation of the electrostatic potential. As one would expect, increasing the number of cells reduces on average the maximum error for the calculation of the potential per atom, and the same for the total error. A maximum error of 0.25% is obtained when we use 35000 cells to compress the whole ligand database. At the same time, memory consumption increases only linearly given the efficiency of the compression method used and only around 300 KB are needed to store the whole ligand database.

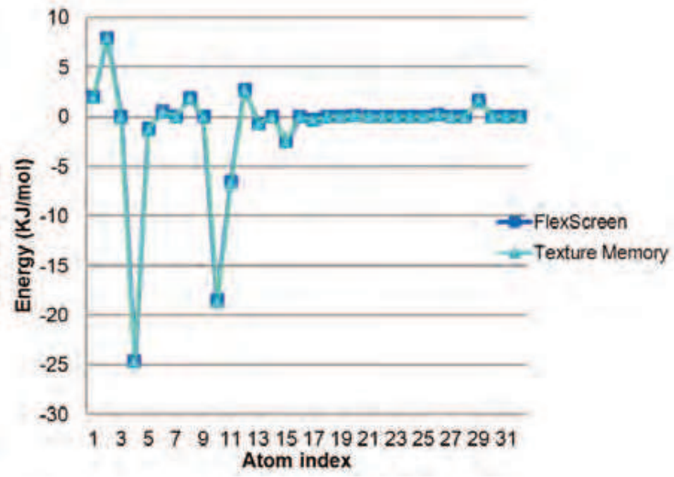


Fig. 12. Value of the electrostatic potential calculated for each atom of biotin in the binding pocket of streptavidin comparing the grid approach used in FlexScreen and using the Texture Memory of the GPU.

4.4.5 Threads per block

We have also investigated the influence of the number of threads per block as can be seen in Figure 11(B). Since the designs ROI-SM and ROI-SM-ZIP use the shared memory, they are more affected by the value of the block size than the others. If the number of threads per block is smaller than the ROI size, threads need to perform too many iterations to copy the whole ROI into the shared memory while the high bandwidth memory is unused. For a number of threads per block higher than the ROI size, the memory access bandwidth is improved because there are many simultaneous memory accesses. The global and texture memory are cached (only in the Fermi architecture) and the data copy is performed automatically in background independently of the number of threads. As a consecuence we have chosen a number of threads equal to 512 as optimal for the shared memory designs.

4.4.6 Texture memory

In the TM strategy we have first checked whether we obtain the same interpolation results than in the sequential version and this is confirmed in Figure 12. We can also see how the use of this memory unit decreases considerably the time needed for the calculation of the interpolation. It is clearly shown in Figure 10(A) in cases GRID-GM to GRID-TM and ROI-SM or ROI-SM-ZIP to ROI-TM. Therefore it is a good idea to use always the TMI when linear interpolation is required. Finally, if we look at Figure 10(A) it is clear that ROI-TM-ZIP and ROI-SM-ZIP offer the best performance since they combine all the best advantages from the previous strategies.

4.4.7 Floating point accuracy influence on different GPUs

We have also performed this study in several NVIDIA GPUs, both in simple and double precision, in order to check how the architectural design affects performance and precision. In Figure 10(B) we can observe that on average, Tesla GPU runs faster than Fermi GPU. For both cases the running times are smaller working on single than in double floating point precision,

as one would expect. In the results obtained in the different GPU strategies presented, Tesla also outperforms Fermi due to the higher number of cores (448 versus 352). This is more accurate in the cases like GRID-GM where interpolation computations take a high percentage of the total running time.

4.5 MURCIA: Implicit solvation and molecular visualization

It is very important in clinical research to determine the safety and effectiveness of current drugs and to accelerate findings in basic research (discovery of new leads and active compounds) into meaningful health outcomes. Both objectives imply to be able to process the vast amount of protein structure data available in biological databases like PDB and also derived from genomic data using techniques as homology modelling (Sanchez & Sali, 1998). Screenings in lab and compound optimization are expensive and slow methods, but bioinformatics can vastly help clinical research for the mentioned purposes by providing prediction of the toxicity of drugs and activity in non-tested targets, by evolving discovered active compounds into drugs for the clinical trials. All this can be done thanks to the availability of bioinformatics tools and Virtual Screening (VS) methods that allow to test all required hypothesis before clinical trials. Nevertheless, VS methods fail to make good toxicity and activity predictions since they are constrained by the access to computational resources; even the nowadays fastest VS methods cannot process large biological databases in a reasonable time-frame. This imposes, thus a serious limitation in many areas of translational research.

We have previously studied how exploitation of last generation massively parallel hardware architectures like GPUs can tremendously overcome this problem accelerating the required calculations and allowing the introduction of improvements in the biophysical models not affordable in the past (Perez Sanchez & Wenzel, 2011). Between the most relevant computationally intensive kernels present in current VS methods, we may highlight the calculation of the molecular surface in terms of the solvent accessible surface area (SASA). We can model efficiently solvation in an implicit way by the calculation of SASA and posterior consideration of the hydrophobic and hydrophilic character of individual atoms (Eisenberg & McLachlan, 1986), being this method widely applied nowadays in protein structure prediction and protein-ligand binding. There have been several efforts to develop a fast method for the SASA calculation. To the best of our knowledge, the fastest method nowadays is POWERSASA (Klenin et al., 2011). Its running time depends linearly on the number of atoms of the molecule. We propose a new method called MURCIA (Molecular Unburied Rapid Calculation of Individual Areas) that uses the GPU as underlying hardware and which runs around 15 times faster than POWERSASA for the usual proteins that we found in most VS methods, with less than 25000 atoms. Another advantage of MURCIA is that it can rapidly provide molecular surface information useful for fast visualization in several molecular graphics programs.

4.5.1 SASA calculation using atomic grids

All atoms of the molecule are specified by their centers and SASA radii, which depend on their Van der Waals radius, and therefore on their atomic type, plus the water molecule radius. MURCIA calculates individual SASA values through the next three Kernels:

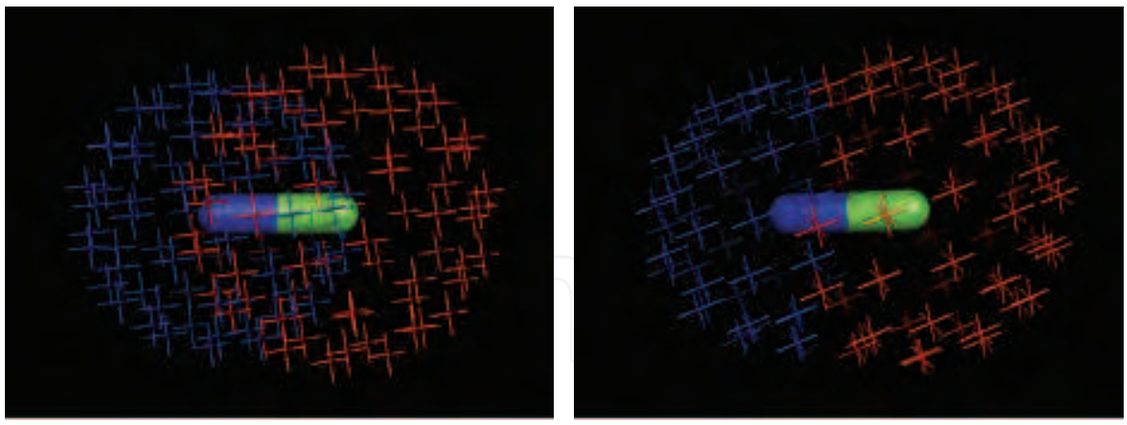


Fig. 13. Atomic grids for a molecule with two atoms (a) both grids overlap, situation previous to the SASA calculation, and (b) only non-buried grid points are shown

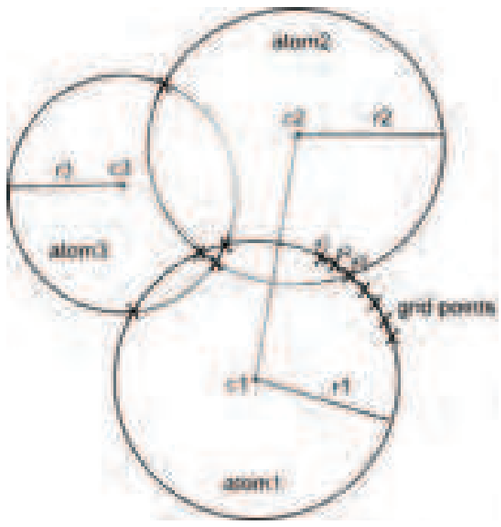


Fig. 14. Depiction in 2D of the SASA calculation in MURCIA

1. *GenGrid*: we build a grid of points around each atom following the procedure developed by Lebedev & Laikov (1999) for the numerical integration over a sphere. This grid guarantees a high precision in integrations, using a very low number of grid points over an unit sphere. In our case we use 72 points. An example of the grid is shown in Figure 13(a).
2. *Neighbours*: we calculate the list of its closest neighbours for each atom. The distance threshold is equal two times the highest value of the highest SASA radii. Atoms are sorted in the lists starting from the closest ones.
3. *Out points*: as depicted in Figure 14 for each atom $i_{,}$ we perform the following calculation for each grid point k; we calculate squared distance to the first neighbour atom j of the list. If this distance is smaller than the SASA radius of atom j, then we flag this grid point as buried. Otherwise we continue the same procedure calculating distances versus the other atoms of the list. If the grid point k is not eventually flagged as buried, then it is stored as contributor to SASA for atom i. Once this procedure is finished for all grid points of

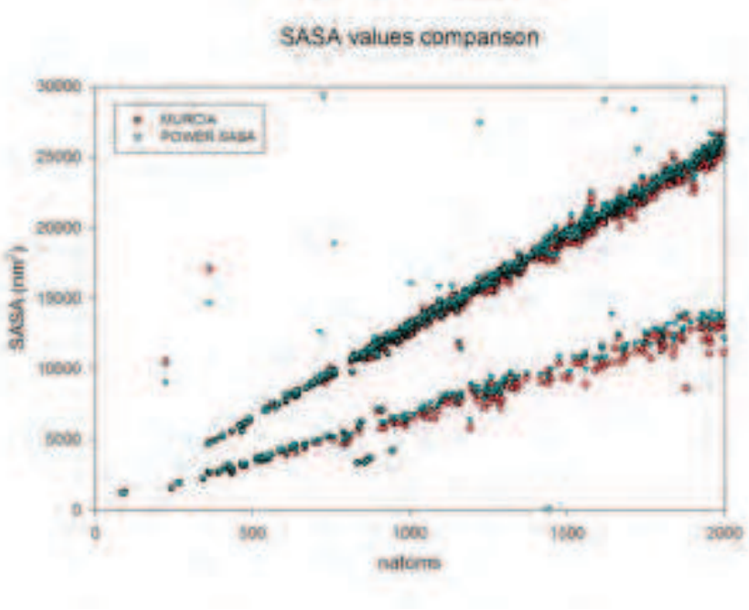


Fig. 15. Comparison of the SASA values calculated by POWERSASA and MURCIA. A diverse set of the PDB database was used for the calculations.

atom i, we will have n non-buried grid points, and individual SASA for this atom will be calculated according to a $(n/72)$ fraction of the sphere surface of radius corresponding to the SASA radius of this atom. At the same time, all coordinates of non-buried grid points are stored for posterior molecular visualization. The same procedure is applied to all atoms i of the molecule. An example of the resulting non-buried grid points is shown in Figure 13(b).

4.5.2 GPU implementation

We used the version 4.0 of the CUDA programming model (NVIDIA, 2011) in our parallel implementation with a NVIDIA Tesla C2050 GPU. In order to obtain speedup measurements versus the sequential counterpart version, an Intel Xeon E5450 cluster was used. This model allows writing parallel programs for GPUs using extensions of the C language. We describe here how the previous kernels are implemented on the GPU:

1. *GenGrid*: It generates atomic grids from the molecular input file. It divides the number of calculation for atoms into CUDA blocks, and assigns a number of threads per block proportional to the number of grid points per atom (72), so each thread computes only one grid point per atom.
2. *Neighbours*: It creates one CUDA block per atom and a variable number of threads per block. Each thread computes for each atom i the distances to the other atoms j. All threads from a block cooperate together to calculate all its neighbours using CUDA shared memory for storing variables commons to all threads of a block.
3. *Out Points*: It establishes the values of number of blocks and threads per block in the same way as the GenGrid kernel does. Each thread computes only distances between only one grid point and all of its neighbours.

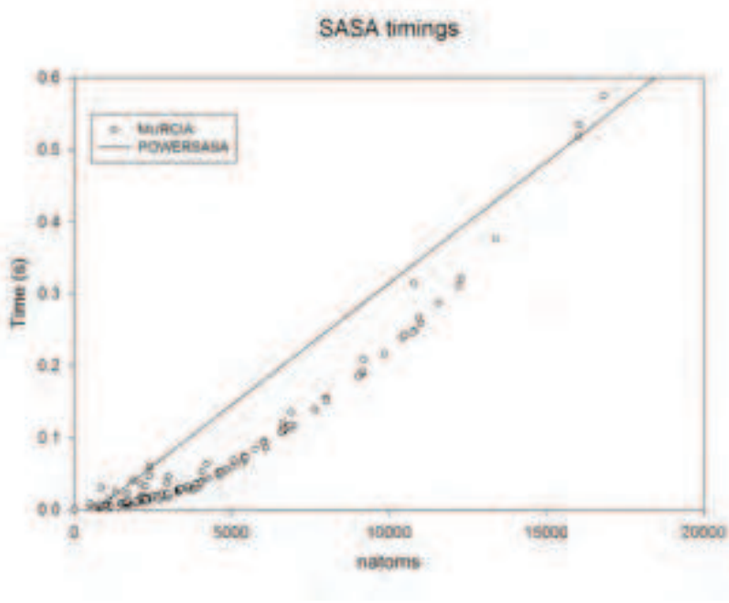


Fig. 16. Comparison of timings for SASA calculation using MURCIA and POWERSASA (since its dependence with the number of atoms is linear and for the sake of clarity, a solid line is used to represent its timings results). A diverse set of the PDB database was used for the calculations.

4.5.3 Performance evaluation

In order to check the accuracy of our method, we check MURCIA calculations with previous POWERSASA results (Klenin 2011). Figure 15 shows an overall good concordance between both methods. POWERSASA uses a very accurate method for the calculation of SASA. There are some cases where MURCIA deviates from the POWERSASA ones. We think this is due to the insufficient number of points (72) used for the atomic grids.

Figure 16 shows a performance comparison between MURCIA and POWERSASA. In the interval 10 to 17000 atoms, MURCIA runs faster than POWERSASA, achieving maximum speedups of 15x. For bigger molecules (20000-100000 atoms) POWERSASA runs faster than MURCIA. We have also checked that MURCIA runs around 30 times faster than MSMS (Sanner et al., 1996).

We have shown in this Section a fast and efficient method for the SASA calculation, implemented on GPU hardware, and which can also be used for fast visualization of molecular surfaces using information calculated for the non-buried atomic surfaces. Nevertheless, the method can be improved since more dense grids influences on the precision of the SASA calculation. Also, the main bottleneck of the program resides in the calculation of neighbours; a better strategy, which calculates much faster the neighbour's list might help considerably. Finally, MURCIA speedups visualization of molecular surfaces in some molecular graphics programs (VMD, Chimera, Pymol).

5. Conclusions and perspectives

After having shown current tendencies in these fields and also our main contributions, we think that the investigations on the improvement of the computational performance of

VS methods on GPUs will be also of high technological interest. Knowledge obtained in the works described here will be transferred to another scientists to port another scientific applications to the next generation of GPUs. So the results obtained in these research lines will offer major technological advantages: (i) performance offered by new parallel architectures will be unveiled; and (ii) cheap hardware equipment will be purchased instead of high-end expensive supercomputers. GPUs are likely to play a very important role in the next generation of VS methods. Thanks to the improvements obtained by implementation and optimization of VS methods in GPUs it will be possible to increase the details in simulations so that more refined and computationally expensive methods will be available at low cost. Supercomputing will be accessible for everyone, and scientific knowledge will advance faster. Techniques such as MD and QM methods, not widely used before due to their high computational cost, will become regular parts of VS methods. Global throughput of VS methods will increase and it will be possible to simulate more and longer trajectories within shorter times. It will be possible to use more accurate thermodynamic methods and compute free energies. Free-energy calculations will move away from individual predictions to form part of high-throughput VS methods.

6. References

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P. & Tomov, S. (2009). Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *Journal of Physics: Conference Series* 180(1): 012037. URL: *http://dx.doi.org/10.1088/1742-6596/180/1/012037*

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), ACM, New York, NY, USA, pp. 483–485. URL: *http://doi.acm.org/10.1145/1465482.1465560*

ATI/AMD (2011). ATI Stream Webpage. `http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/st%ream-technology.aspx`.

Bohn, C.-A. (1998). Kohonen Feature Mapping through Graphics Hardware, *Proceedings of International Conference on Computational Intelligence and Neurosciences*, ICCIN 98, pp. 64–67.

Bursulaya, B. D., Totrov, M., Abagyan, R. & Brooks, C. L. (2003). Comparative study of several algorithms for flexible ligand docking., *J Comput Aided Mol Des* 17(11): 755–763. URL: *http://view.ncbi.nlm.nih.gov/pubmed/15072435*

Cecilia, J. M., García, J. M., Ujaldón, M., Nisbet, A. & Amos, M. (2011). Parallelization strategies for ant colony optimisation on gpus, *NIDISC '2011: 14th International Workshop on Nature Inspired Distributed Computing. Proc. 25th International Parallel and Distributed Processing Symposium (IPDPS 2011)*, Anchorage (Alaska), USA.

Cerutti, D. S., Duke, R. E., Darden, T. A. & Lybrand, T. P. (2009). Staggered Mesh Ewald: An Extension of the Smooth Particle-Mesh Ewald Method Adding Great Versatility, *Journal Of Chemical Theory And Computation* 5(9): 2322–2338.

CUD (2011). The CUDA Zone website. `http://www.nvidia.com/object/cuda_home_new.html`.

Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy.

Eisenberg, D. & McLachlan, A. D. (1986). Solvation energy in protein folding and binding., *Nature* 319(6050): 199–203.

England, J. N. (1978). A System for Interactive Modeling of Physical Curved Surface Objects, *Proceedings of the 5th annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 78, ACM, pp. 336–340.

Feng, Z.-w., Tian, X.-h. & Chang, S. (2010). A Parallel Molecular Docking Approach Based on Graphic Processing Unit, *Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on*, pp. 1–4.

Fischer, B., Basili, S., Merlitz, H. & Wenzel, W. (2007). Accuracy of binding mode prediction with a cascadic stochastic tunneling method, *Proteins: Structure, Function, and Bioinformatics* 68(1): 195–204.

Friesner, R. A. & Banks, J. L. (2004). Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy, *Journal of Medicinal Chemistry* 47(7): 1739–1749.
URL: *http://dx.doi.org/10.1021/jm0306430*

Garland, M. & Kirk, D. B. (2010). Understanding throughput-oriented architectures, *Commun. ACM* 53: 58–66.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. & Volkov, V. (2008). Parallel computing experiences with cuda, *IEEE Micro* 28: 13–27.

Graham, S. L., Kessler, P. B. & McKusick, M. K. (2004). gprof: a call graph execution profiler, *SIGPLAN Not.* 39: 49–57.
URL: *http://doi.acm.org/10.1145/989393.989401*

Guerrero, G., Pérez-Sánchez, H., Wenzel, W., Cecilia, J. M. & García, J. M. (2011). Effective parallelization of non-bonded interactions kernel for virtual screening on gpus, *5th International Conference on Practical Applications of Computational Biology; Bioinformatics (PACBB 2011)*, Vol. 93, Springer Berlin / Heidelberg, pp. 63–69.

Harris, M. J. (2002). Analysis of Error in a CML Diffusion Operation, *Technical Report TR02-015*, University of North Carolina.

Harvey, M. J. & De Fabritiis, G. (2009). An Implementation of the Smooth Particle Mesh Ewald Method on GPU Hardware, *Journal of Chemical Theory and Computation* 5(9): 2371–2377.
URL: *http://dx.doi.org/10.1021/ct900275y*

Hetényi, C. & van der Spoel, D. (2002). Efficient docking of peptides to proteins without prior knowledge of the binding site., *Protein science : a publication of the Protein Society.* 11(7): 1729–1737.
URL: *http://dx.doi.org/10.1110/ps.0202302*

Hoff, III, K. E., Zaferakis, A., Lin, M. & Manocha, D. (2001). Fast and simple 2D geometric proximity queries using Graphics hardware, *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D 01, ACM, pp. 145–148.

Hopf, M. & Ertl, T. (1999). Accelerating 3D convolution using Graphics hardware (case study), *Proceedings of the Conference on Visualization*, VIS 99, IEEE Computer Society Press, pp. 471–474.

Hopf, M. & Thomas, T. (1999). Hardware Based Wavelet Transformations, *Proceedings of Workshop on Vision, Modeling, and Visualization*, pp. 317–328.

Hui-fang, L., Qing, S., Jian, Z. & Wei, F. (2010). Evaluation of various inverse docking schemes in multiple targets identification., *Journal of molecular graphics & modelling* 29(3): 326–330.

Hwu, W.-m. W. (ed.) (2011). *GPU Computing Gems: Emerald Edition*, Morgan Kaufmann.

Jorgensen, W.L., M. D. T.-R. J. (1996). Development and testing of the opls all-atom force field on conformational energetics and properties of organic liquids, *Journal of the American Chemical Society* 118(45): 11225–11236.

Kannan, S. & Ganji, R. (2010). Porting Autodock to CUDA, *Evolutionary Computation (CEC), 2010 IEEE Congress on* pp. 1–8.

Kedem, G. & Ishihara, Y. (1999). Brute force attack on UNIX passwords with SIMD Computer, *Proceedings of the 8th Conference on USENIX Security Symposium*, USENIX Association, pp. 8–8.

Kirk, D. B. & Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann.

Klenin, K. V., Tristram, F., Strunk, T. & Wenzel, W. (2011). Derivatives of molecular surface area and volume: Simple and exact analytical formulas., *Journal of Computational Chemistry* 32(12): 2647–2653.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. & Fasih, A. (2011). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation, *ArXiv e-prints* .
URL: *http://arxiv.org/abs/0911.3456*

Kokh, D. B., Wade, R. C. & Wenzel, W. (2011). Receptor flexibility in small-molecule docking calculations, *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1(2): 298–314.
URL: *http://dx.doi.org/10.1002/wcms.29*

Kokh, D. B. & Wenzel, W. (2008). Flexible side chain models improve enrichment rates in in silico screening, *Journal of Medicinal Chemistry* 51(19): 5919–5931.

Korb, O., Stützle, T. & Exner, T. (2006). PLANTS: Application of Ant Colony Optimization to Structure-Based Drug Design, *in* M. Dorigo, L. Gambardella, M. Birattari, A. Martinoli, R. Poli & T. Stützle (eds), *Ant Colony Optimization and Swarm Intelligence*, Vol. 4150 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, Berlin, Heidelberg, chapter 22, pp. 247–258.

Korb, O., Stützle, T. & Exner, T. E. (2011). Accelerating molecular docking calculations using graphics processing units., *Journal of chemical information and modeling* 51(4): 865–876.

Larsen, E. S. & McAllister, D. (2001). Fast matrix multiplies using Graphics hardware, *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Supercomputing 01, ACM, pp. 55–55.

Lebedev, V. I. & Laikov, D. N. (1999). A quadrature formula for the sphere of the 131st algebraic order of accuracy, *Doklady Mathematics* 59(3): 477–481.

Lengyel, J., Reichert, M., Donald, B. R. & Greenberg, D. P. (1990). Real-time robot motion planning using rasterizing computer graphics hardware, *Proceedings of the 17th annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 90, ACM, pp. 327–335.

Lindholm, E., Kilgard, M. J. & Moreton, H. (2001). A user-programmable vertex engine, *Proceedings of the 28th annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 01, ACM, pp. 149–158.

Luebke, D. (2007). The Democratization of Parallel Computing. Keynote at International Conference on Supercomputing.

Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M. & Buck, I. (2006). GPGPU: General-Purpose Computation on Graphics hardware, *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC 2006, ACM.

Manfrin, M., Birattari, M., Stützle, T. & Dorigo, M. (2006). Parallel ant colony optimization for the traveling salesman problem, *in* M. Dorigo, L. M. Gambardella, M. Birattari, A. Martinoli, R. Poli & T. Stützle (eds), *Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS˜2006*, Vol. 4150 of *LNCS*, Springer Verlag, Berlin, Germany, pp. 224–234.

Meliciani, I., Perez Sanchez, H. & Wenzel, W. (2009). Analysis of the complex antithrombin/thrombin and alanine mutation of the complex antithrombin/heparin using two different docking approaches (poem, flexscreen).

Meng, E., Shoichet, B. & Kuntz, I. (1992). Automated Docking with Grid-Based Energy Evaluation, *Journal of Computational Chemistry* 13(4): 505–524.

Merlitz, H., Burghardt, B. & Wenzel, W. (2003). Application of the stochastic tunneling method to high throughput database screening, *Chemical physics letters* 370(1-2): 68–73.

Merlitz, H., Herges, T. & Wenzel, W. (2004). Fluctuation analysis and accuracy of a large-scalein silico screen, *Journal of Computational Chemistry* 25(13): 1568–1575.

Navarro-Fernandez, J., Martinez-Martinez, I., Perez-Sanchez, H., Meliciani, I., Wenzel, W., de la Morena-Barrio, M., Vicente, V. & Corral, J. (2010). Identification of a compound with enhanced capacity in the activation of antithrombin in presence of heparin.

Nguyen, H. (2007). *GPU Gems 3*, Addison-Wesley Professional.

NVIDIA (2009). *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi*.

NVIDIA (2010). *NVIDIA CUDA Programming Guide 4.0*.

NVIDIA (2011). *NVIDIA CUDA C Programming Guide 4.0*.

Owens, John, D., Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, Aaron, E., Purcell & Timothy, J. (2007). A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum* 26(1): 80–113.

Pérez-Sánchez, H. (2009). Implementation of an Effective Non-Bonded Interactions Kernel for Biomolecular Simulations on the Cell Processor, *Gesselschaft für Informatik* .

Perez Sanchez, H. & Wenzel, W. (2011). Optimization methods for virtual screening on novel computational architectures., *Current computer-aided drug design* 7(1): 44–52.

Pharr, M. & Fernando, R. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional.

Potmesil, M. & Hoffert, E. M. (1989). The Pixel Machine: A Parallel Image Computer, *Proceedings of the 16th annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 89, ACM, pp. 69–78.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edn, Cambridge University Press, New York, NY, USA.

Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. & Varshney, A. (1992). Real-time procedural textures, *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, I3D 92, ACM, pp. 95–100.

Roh, Y., Lee, J., Park, S. & Kim, J.-I. (2009). A molecular docking system using CUDA, *ICHIT '09: Proceedings of the 2009 International Conference on Hybrid Information Technology* .

Sanchez, R. & Sali, A. (1998). Large-Scale Protein Structure Modeling of the Saccharomyces cerevisiae Genome, *Proceedings Of The National Academy Of Sciences Of The United States Of America* 95(23): 13597–13602.

Sanders, J. & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional.

Sanner, M. F., Olson, A. J. & Spehner, J. C. (1996). Reduced surface: an efficient way to compute molecular surfaces., *Biopolymers* 38(3): 305–320.

Schneider, G. & Böhm, H. J. (2002). Virtual screening and fast automated docking methods., *Drug Discov Today* 7(1): 64–70.
URL: *http://view.ncbi.nlm.nih.gov/pubmed/11790605*

Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G. & Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry* 28(16): 2618–2640.

Strzodka, R. (2002). Virtual 16 Bit Precise Operations an RGBA8 Textures, *Proceedings of the Vision, Modeling, and Visualization Conference*, VMV 2002, pp. 171–178.

Strzodka, R. (2004). *Hardware Efficient PDE Solvers in Quantized Image Processing*, PhD thesis, University of Duisburg-Essen.

Sukhwani, B. & Herbordt, M. (2010). Fast binding site mapping using GPUs and CUDA, *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* pp. 1–8.

Thompson, C. J., Hahn, S. & Oskin, M. (2002). Using modern graphics architectures for general-purpose computing: a framework and analysis, *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, IEEE Computer Society Press, pp. 306–317.

Trendall, C. & Stewart, A. J. (2000). General calculations using graphics hardware with applications to interactive caustics, *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Springer-Verlag, pp. 287–298.

Volkov, V. & Demmel, J. W. (2008). Benchmarking gpus to tune dense linear algebra, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, IEEE Press, Piscataway, NJ, USA, pp. 31:1–31:11.
URL: *http://portal.acm.org/citation.cfm?id=1413370.1413402*

Volkov, V. & Kazian, B. (2008). Fitting fft onto the g80 architecture, *Methodology* p. 6.

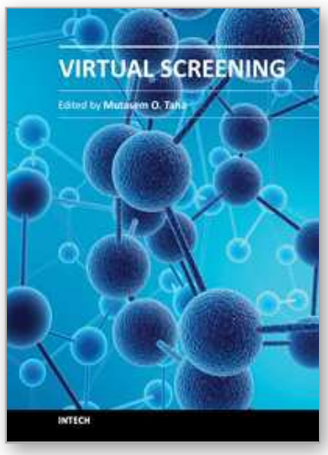
Wang, J., Deng, Y. & Roux, B. (2006). Absolute Binding Free Energy Calculations Using Molecular Dynamics Simulations with Restraining Potentials, *Biophys. J.* 91(8): 2798–2814.
URL: *http://dx.doi.org/10.1529/biophysj.106.084301*

Xing, L., Hodgkin, E., Liu, Q. & Sedlock, D. (2004). Evaluation and application of multiple scoring functions for a virtual screening experiment, *Journal of Computer-Aided Molecular Design* 18(5): 333–344.

Yang, H., Zhou, Q., Li, B., Wang, Y., Luan, Z., Qian, D. & Li, H. (2010). GPU Acceleration of Dock6's Amber Scoring Computation, *Advances in Computational Biology* 680: 497–511.

Yuriev, E., Agostino, M. & Ramsland, P. A. (2011). Challenges and advances in computational docking: 2009 in review, *Journal Of Molecular Recognition* 24(2): 149–164.

Zhang, C., Liu, S., Zhu, Q. & Zhou, Y. (2005). A knowledge-based energy function for protein-ligand, protein-protein, and protein-DNA complexes., *Journal of Medicinal Chemistry* 48(7): 2325–2335.



Virtual Screening
Edited by Prof. Mutasem Taha

ISBN 978-953-51-0308-0
Hard cover, 100 pages
Publisher InTech
Published online 14, March, 2012
Published in print edition March, 2012

Pharmacophore modeling, QSAR analysis, CoMFA, CoMSIA, docking and molecular dynamics simulations, are currently implemented to varying degrees in virtual screening towards discovery of new bioactive hits. Implementation of such techniques requires multidisciplinary knowledge and experience. This volume discusses established methodologies as well as new trends in virtual screening with aim of facilitating their use in drug discovery.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Horacio Pérez-Sánchez, José M. Cecilia and José M. García (2012). Recent Advances and Future Trend on the Emerging Role of GPUs as Platforms for Virtual Screening-Based Drug Discovery, Virtual Screening, Prof. Mutasem Taha (Ed.), ISBN: 978-953-51-0308-0, InTech, Available from: http://www.intechopen.com/books/virtual-screening/exploitation-of-virtual-screening-methods-on-gpus

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen