GPU Accelerated Molecular Docking Simulation with Genetic Algorithms

Serkan Altuntaş, Zeki Bozkus and Basilio B. Fraguel[1]
Department of Computer Engineering, Kadir Has Üniversitesi, Turkey,
serkan.altuntas@stu.khas.edu.tr, zeki.bozkus@khas.edu.tr
[1]Depto. de Electrónica e Sistemas, Universidade da Coruña, Spain
basilio.fraguela@udc.es

Abstract. Receptor-Ligand Molecular Docking is a very computationally expensive process used to predict possible drug candidates for many diseases. A faster docking technique would help life scientists to discover better therapeutics with less effort and time. The requirement of long execution times may mean using a less accurate evaluation of drug candidates potentially increasing the number of false-positive solutions, which require expensive chemical and biological procedures to be discarded. Thus the development of fast and accurate enough docking algorithms greatly reduces wasted drug development resources, helping life scientists discover better therapeutics with less effort and time.

In this article we present the GPU-based acceleration of our recently developed molecular docking code. We focus on offloading the most computationally intensive part of any docking simulation, which is the genetic algorithm, to accelerators, as it is very well suited to them. We show how the main functions of the genetic algorithm can be mapped to the GPU. The GPU-accelerated system achieves a speedup of around ~14x with respect to a single CPU core. This makes it very productive to use GPU for small molecule docking cases.

Keywords: GPU · OpenCL · molecular docking · genetic algorithm · parallelization

1. Introduction

The binding of small molecule ligands to large protein targets is central to numerous biological processes. For example the accurate prediction of the binding modes between a ligand and a protein, which is known as the docking problem, is of fundamental importance in modern structure-based drug design [1]. Docking algorithms try to generate different poses (binding modes) throughout possible three-dimensional conformations, which can be seen in Figure 1, with the purpose of evaluating them so as to choose the best possible pose.

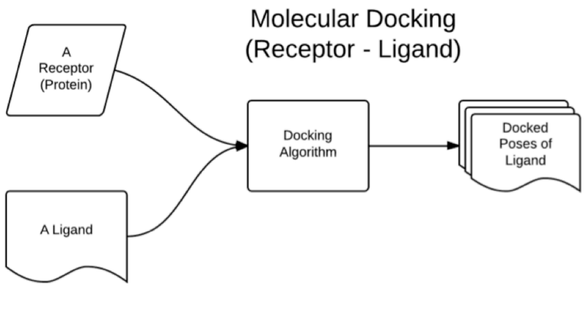

Fig 1. Molecular Docking (Receptor - Ligand)

Molecular dynamic applications use a highly sophisticated force field while searching only a small portion of the conformational space. This approach uses physically based energy functions combined with full atomic level simulations to yield accurate estimates of the energetics of molecular processes. However, these methods are too computationally time consuming to allow blind docking of a ligand to a protein. On the other hand, molecular docking simulations often use simpler force fields and explore a wider region of the conformational space [2].

Docking simulations require two basic components:

- A search method for exploring the conformational space available to the system.
- A force field to evaluate the energetics of each conformation (scoring function).

The extensive search performed by docking algorithms involves the sampling of many high-energy unfavorable states. This can restrict the success of an optimization algorithm. Therefore the computational expense is limited by applying constraints, restraints and approximations to sample such a large search space. In practice a limited search may reduce the dimensionality of the problem in an attempt to locate the global minimum as efficiently as possible [1].

Stochastic methods such as Monte Carlo (MC) or genetic algorithms (GA) are general optimization techniques with a limited physical basis, and are able to explore the search. Evolutionary algorithms are generic iterative stochastic optimization procedures mimicking the adaptive process of natural evolution, classified as artificial intelligence techniques [3].

We develop a docking algorithm whose search method is a GA. Based on genotypes of parents. Each generation has a number of offspring, which replaces the worst solutions of the population since population size is limited. The latter is then exposed again to the scoring function, and the evolution goes on for the next iteration (generation). As the number of generations increases, the average fitness of the population of solutions is supposed to increase, and several highly fit solutions are expected to appear.

The fact that almost all subunits of GAs have an embarrassingly parallel nature makes these algorithms very suitable for massively parallel accelerators such as graphics processing units (GPUs). While GPUs were originally designed to process graphics with a massive number of threads, in recent years, they have evolved to accelerate broader types of applications. This has been favored by the availability of new programming models such as Compute Unified Device Architecture (CUDA) [4] and Open Computing Language (OpenCL) [5], which allow the development of general purpose, non graphics programs for GPUs. We have implemented our application using the Heterogeneous Programming Library (HPL) [6], [7] because it largely improves the programmability of heterogeneous platforms with respect to CUDA and OpenCL, while it presents negligible performance overheads. HPL works on top of OpenCL, the standard for heterogeneous computing, which guarantees that our application can be used in a wide range of devices.

Our heterogeneous implementation prepares in the host the receptor binding site and ligand conformation and then it transfers it to the accelerator (the GPU). This device then creates the first population and starting from this point to the final result, all the GA operations are computed on the GPU. The design is a very important contribution of our work, as the fact that there is no CPU involvement during the execution of the GA, so that everything is locally computed in the GPU, avoids costly data movements between CPU and GPU, which would affect performance negatively. Following this strategy our implementation achieved in our tests average speedups of around 14x with respect to a single core CPU. We have also analyzed the reason for the performance obtained, identifying a several memory optimizations to improve the performance even further.

2. Docking Algorithm

Small molecule docking is the algorithm to predict the preferred binding pose of a small molecule to a target molecule (a protein). Small molecules are known as *ligands* and they may have different *poses,* which are different orientations on 3D space of the same chemical component. These orientations can be stored with the coordinates of an indicator atom and *torsion angles* of ligand molecule. Torsion angles represent the rotation between two imaginary geometric planes of the molecule. The number of torsions is known as the *torsion size* and it is fixed on all poses.

The receptor has many atoms that are not related to the docking of the ligand. This way, only a part of the receptor atoms interact with the ligand, and for the sake of fast execution docking algorithms target directly to a local site on the molecule. This *site* (also called; *binding site*, *binding pocket*, *binding cavity*) is decided by the user and defined as a sphere by a center point and a radius. If the binding site covers all receptor (this means that the user does not have information about the specific binding site) the docking is called *blind docking*. This kind of docking algorithm should find the pocket first. As a result blind docking requires more GA runs and uses all the atoms of receptor.

Molecular Docking consists of three basic steps, which are seeding, selection and diversity. In general both MC and GA based molecular docking simulations perform these three basic steps with different styles. We now briefly describe these three steps in turn:

2.1. Seeding

Decoys from the reference coordinates of the ligand populate the first population from generation zero. These decoys are called as seeds and each one of them is generated by means of a random rotation and translation beginning from the reference coordinates [3].

2.2. Selection

After the creation of the first generation, the scoring function is applied to all the individuals. Then, the parents of the next generation and the leaving individuals are identified based on the rankings obtained.

The scoring function is important for the accuracy of the docking algorithm. Unfortunately, its complexity can largely increase run-time. If we are doing the virtual screening of millions of compounds, we can employ a lightweight scoring function for faster turnaround. On the other hand, some docking algorithm employs adaptive scoring techniques such as changing the complexity of the functions in the last iterations of the simulation [3].

2.3. Diversity

In order to generate a child, two parents are randomly chosen according to their rank after the selection step. Although the features of each child come from its parents and yet this step can also create new features by adjusting valence angles and bond lengths [3].

3. GPU Parallelization: Algorithms and Implementation

Figure 2 describes the docking software architecture, which should be executed for each receptor ligand complex. The figure is simplified to represent only the single docking experiment but the actual preparation of the receptor and the ligand requires some more operations according to the docking software.

3.1 Overview of Data Structure

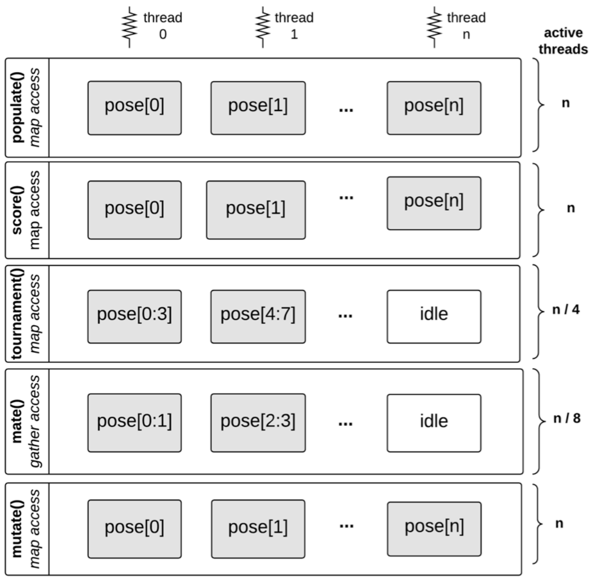

Fig. 2. Threads & Memory Access Patterns

Our docking simulation is based on a genetic algorithm for its *conformational search*. This particular conformational search is the process to find the best possible pose with respect to the receptor. The *pose* is the single individual which includes the 3D conformation of a molecule. There are different properties of each pose like *atom types*, *atom coordinates*, *number of branches*, *branch values* and *calculated scores*. These properties are stored inside multiple arrays that are referred as **property arrays of individuals (PAI)** in this article.

Algorithms and Memory Access Patterns

Our program has two parts. One part runs on CPU and we refer to it as the host program. The other part runs on a GPU and we call it the kernel program, so that our application applies heterogeneous programming.

ALGORITHM 1: PSEUDO-CODE FOR THE HOST SIDE:

```
conf = read_conf()
receptor = read_receptor(conf.receptor_path)
ligand = read_ligand(conf.ligand_path)

// trim rest of the receptor,
// only binding site remains
site = prepare_binding_site(conf, receptor)

// move ligand into the center of binding site
initial = move_ligand_into_site(conf, ligand)

// genetic algorithm for conformational search
result = dock(conf, site, initial)  // GPU accel.

// create the pdb file of best results
write_pdbs(result)
```

Algorithm 1 presents the pseudo-code of our host program. The host code first reads all the required files and prepares with them the required data structures for the receptor and ligand. Then the unused parts of receptor are trimmed, the binding site is generated, and the movement of initial pose is carried out in order to make possible the score calculation.

ALGORITHM 2: PSEUDO-CODE FOR THE GA:

```
// get receptor <receptor>,
// ligand <ligand>,
// configuration parameters <conf>
population = populate(ligand)

// loop: with number of generations
foreach generation from conf {

  // energy calculation
  score(population, receptor)

  // selection
  tournament(population)

  // mate the winners and
  // overwrite 3/4 of population
  mate(population)

  // mutate
  mutate(population)
 }
```

Algorithm 2 presents the kernel portion of our GPU program which is the most computationally intensive part and which implements the conformational search. This algorithm consists of five different main subroutines, all of which are offloaded to the accelerator device. Our GA iterates and terminates according to a predefined number of generations, which is set by user. Each subroutine of the GA has slightly different memory access patterns such as map and gather. The map access pattern reads and writes data following a regular fashion. On the other hand the gather access reads and writes following an irregular fashion that restricts the memory bandwidth of the GPU. Figure 2 summarizes these patterns. We now briefly describe in turn the five stages of the kernel program of our application.

The number of threads used in the kernel executions is relative to the population size (number of individuals) so the global thread domain size is equal to population size. Since the algorithm has been optimized the algorithm to take advantage of local work groups the local domain size is 1.

populate:

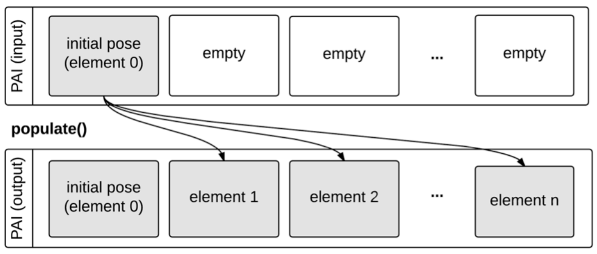

Fig. 3. Data Flow for populate

The first step of the conformational search is the generation of a population of a fixed size. Until this point, there is only one pose of the initial ligand, which is specifically moved into the binding site. It is identified as the 0^{th} element of the property array of individuals that represents the whole population.

In this kernel, each thread t reads the 0^{th} element of the array as input, creates a slightly different variation of it and writes the output to the t-th position of the property arrays. Figure 3 describes the map access type data movement of this step, which is performed only once, just before the simulation loop.

score:
The genetic algorithm relies on the fitness function to take its decisions and find the best fitting individuals. Here the fitness function is the scoring function of the molecular docking.

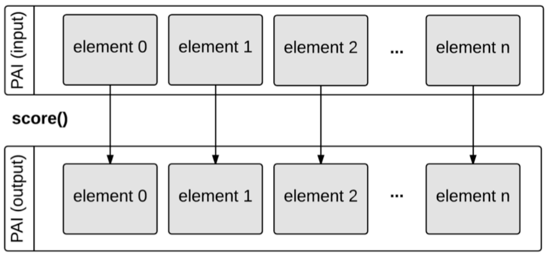

Fig. 4. Data flow for score

The score of each pose is calculated a different thread in a map access fashion, as shown in Figure 4, calculates the score of each pose. The scoring function also uses the binding pocket data, which is the same for all the individuals during the simulation.

tournament:

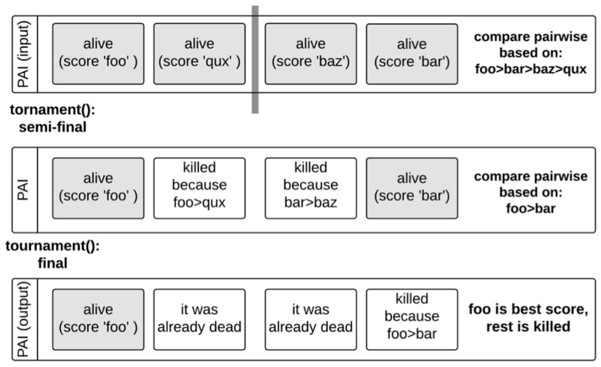

Fig. 5. Data Flow for tournament

The tournament selection is designed to terminate the underperforming individuals in order to create space for new generations. The population is divided into groups of four individuals. The selection method has two stages: semi-final and final. The semi-final stage compares the score of two consecutive individuals and eliminates the worst half. Then, the final competition chooses a single winner for each group after comparing the results of the semi-final stage. Figure 5 represents this process. As we can see this method cannot use all the available GPU threads, which is a problem that is typical of reduction processes. Namely, the semi-final stage uses half of the threads, while the final stage only uses a quarter of all threads.

mate:

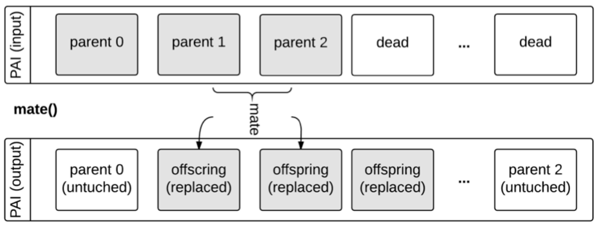

Fig. 6. Data Flow for mate

The tournament process eliminates ¾ of all the individuals. The mating stage should fill the empty spaces because the GA relays on a fixed population size. In this process, illustrated in Figure 6, each thread randomly picks two parents among the alive individuals, produces new offsprings from the chosen parents, and writes them to the available spaces, marked as empty. The user sets a *crossover ratio,* which is a floating-point number between 0 and 1. Based on this ratio some offsprings will have parts from both parents and some may be exact copies of one of their parents. Although mating is the only process that produces newly formed individuals, crossover is applied by chance.

mutate:

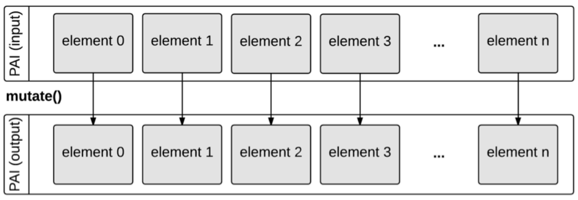

Fig. 7. Data Flow for mutate

Mutation is a key part of GA because it is the main source of biologically inspired variation. In this stage of the algorithm each thread gets an individual and makes variations based on a variation ratio, which is set by user. Figure 7 represents the map access for mutate function.

4. Performance Results

In this section we show the results of our experiments to compare the performance results of serial and parallel GA. We used a Tesla C2050/C2070 GPU as experimental platform. The device has 448 thread processors with a clock rate of 1.15 GHz and 6GB of DRAM and it is connected to a host system consisting of 4x Dual-Cores Intel 2.13 GHz Xeon processors. The compiler used for all tests was g++ 4.7.1 with optimization level O3. This compiler supports C++11 standards, which is necessary for our code.

Figure 8 represents the execution time over a varying number of GA runs for a single docking performs docking between a ligand of 25 atoms and a large receptor. In our docking algorithm, we choose a binding site from the receptor with 528 atoms.

Users of docking programs (i.e. AutoDock) prefer at least 10 GA runs to make sure the results are satisfying. The reason is that although every GA run includes a full population and iterates for the number of generations specified by the user, a single GA run is not enough for finding a good enough pose because GAs are easy to stuck on local solutions.

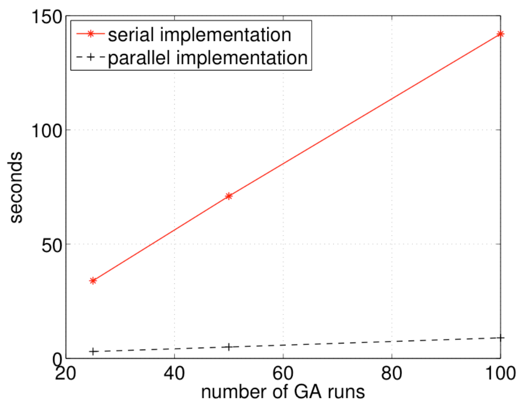

Fig. 8. Execution time over number of GA runs

We have performed our speedup tests using three different compounds (A, B, C). Each one has a different torsion size (7, 5, 8) and a different number of atoms (25, 19, 28). As Table 1 and Figure 9 represent, the compound size (number of atoms and number of torsions) does not have any important effect on the speedup, but the number of GA runs makes a difference.

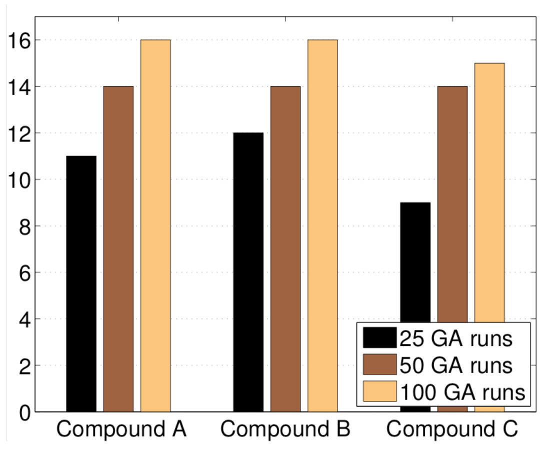

Fig. 9. Speedup on Tesla C2050/C2070 GPU for Docking Algorithm

The main reason for the increasing speedup as the number of runs grows is the initialization cost of the HPL/OpenCL framework. In addition, our program spends a fixed amount of time for preparing the receptor and the ligand during a single docking experiment. That is, at the beginning of the first simulation there is a one time cost for the whole docking simulation, which is the same no matter how many GA runs will be performed.

TABLE I. SPEEDUP

	Speedup		
	25 GA runs	***50 GA runs***	***100 GA runs***
Compound A	11	14	16
Compound B	12	14	16
Compound C	9	14	15

We profiled our GPU code for performance bottlenecks. This analysis revealed that our algorithm is not fully using the memory bandwidth of GPU. We concluded that our implementation needs two kind different memory optimizations. The first approach is to use tiling with the shared memory of the GPU streaming multi processors (SM) and in general reconsider the kind of memory used for our data. For example the receptor binding site data is currently accessed from global device memory but this data never changes during the computation, so the memory type for this data should be reconsidered. The second important optimization is to transform the data layout in order to increase the number of coalesced memory access to global memory at the GPU. Namely our code currently stores its data using arrays of structures (AOS). Changing this layout to use structures of arrays (SAO) will allow improving the bandwidth of our accesses to the global memory.

5. Related Work

AutoDock is one of the best-known docking applications. In fact it was the most cited docking program in the ISI Web of Science database in 2005 [8], [9]. In addition to single CPU docking implementations, many research groups have created GPU and FPGA based solutions.

GPU Based Molecular Docking Implementations

Micevski D. et al, [10], [11] profiled the original AutoDock code and identified two different functions (eintcal and trilininterp) that were suitable to be ported to GPU. Each time these functions are called in their version the corresponding CUDA kernel is executed instead of the original function. In both cases the number of threads within the kernel equals to the number of ligand atoms. This gives place to a very low utilization of the GPU device.

Kannan S. et al, [11], [12] reported the migration of AutoDock to NVIDIA CUDA with the only exception of the local search part, since genetic algorithms are relatively straightforward to be ported to GPU platforms. Their implementation keeps the storage of the ligand coordinates in fast shared memory, which makes the score evaluation faster. For determining the atom-receptor intermolecular energy, each thread first performs a trilinear interpolation for a different ligand atom, and then each thread evaluates the scoring function directly for a different ligand-ligand atom pair. Trilinear interpolation offers a further optimization, since NVIDIA GPUs support the fast access of 3D data by hardware. As a result of this they end up with ~50x speedup on fitness function evaluation and 10x to 47x speedup on the core genetic algorithm.

Unfortunately CUDA is a vendor specific model to address parallel execution that is well suited to NVIDIA GPUs, but it is not portable to other architectures. Our implementation however is based on the Heterogeneous Programming Library (HPL) [6], [7], which creates OpenCL code on runtime. Since OpenCL is the portable standard for heterogeneous computing, we are able to run our code on any OpenCL enabled CPU/GPU from different vendors, and other co-processors like Xeon Phi.

Local search is an important factor for the accuracy of AutoDock [13], and performing it on the CPU gives place to underutilization of GPU resources. Pechan I. et al, [11], [14] introduced two different kernels to add local search functionality. Namely one of them creates and evaluates a whole population and the other one performs the local search. They share the same scoring function but they differ in the calculation of the degree of freedom. As a result of that, they achieved speedups of x30 and x64 with respect to CPU executions, when performing 10 and 100 independent runs, for a large set of ligands, respectively.

FPGA Based Molecular Docking Implementations

Pechan I. et al, [15] targeted the parallel execution of distinct docking runs, which is an obvious approach since there is no relation between different docking runs. Also they evaluated different entities of the same population simultaneously, which resulted in remarkably high performance. Their implementation applies pipelines and fine-grained parallelization, and achieves x10-40 speedup over a 3.2Ghz CPU. In addition to this, they manipulated the scoring function in order to fit it better to FPGA devices. Namely instead of computing the scoring function with floating point precision, they used fixed-point arithmetic, as it is likely that the performance of the algorithm does not decrease with this change, while it fits better the capabilities of an FPGA.

VanCourt T. et al, [16] used a 3D correlation method on FPGA devices. Their approach, based on direct summation, allows straightforward combination of multiple forces and enables nonlinear force models. The latter, in particular, are incompatible with the transform-based techniques typically used.

MPI Based Molecular Docking Implementations

Zhang X. et al, [17] created an MPI and multithreading hybrid which is a task parallel solution of the AutoDock Vina [18]. The aim of their research was to develop having a faster virtual screening result, so they did not report the improvement of single docking performance.

6. Conclusions and Future Work

We describe a GPU-accelerated molecular docking code with a genetic algorithm. Our code achieves a speedup of around 14x with respect to a single core. We found that our code is not fully using the memory bandwidth of GPU system, a critical reason for this being that our current data layout is not well suited for the GPU and the special kinds of memories in GPUs, such as local memory, are not fully exploited. Despite this fact, the speedup achieved indicates that the GPU architecture is one of the best possible solutions for GA implementations.

An interesting property of our implementation of molecular docking GA is that it is specifically developed for full GPU-based computation, being the host code just a wrapper for the I/O requirements of our software.

Our work only includes single device executions of independent GA runs for a specified number of iterations. This makes the problem very suitable for multi device implementation, which is part of our future work.

Finally, our GPU implementation of docking makes single docking fast but designing the algorithm for virtual screening of drug candidates, which is another important topic for molecular docking, may lead to better results.

7. Acknowledgment

Sekan Altuntaş and Zeki Bozkus are funded by the Scientific and Technological Research Council of Turkey (TUBITAK; 112E191). Basilio B. Fraguela is supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (ref. TIN2013-42148-P) and by the Galician Government under the Consolidation Program of Competitive Reference Groups (ref. GRC2013- 055).

8. Reference

[1] R. D. Taylor, P. J. Jewsbury, and J. W. Essex, “A review of protein-small molecule docking methods.,” *J. Comput. Aided. Mol. Des.*, vol. 16, no. 3, pp. 151–166, Mar. 2002.

[2] R. Huey, G. M. Morris, A. J. Olson, and D. S. Goodsell, “Software News and Update A Semiempirical Free Energy Force Field with Charge-Based Desolvation,” 2007.

[3] A. Grosdidier, "EADock: Design of a New Molecular Docking Algorithm and Some of Its Applications," 2007.

[4] R. Manual, "Compute Unified Device Architecture," *J. Inst. Image Inf. Telev. Eng. ITE*, vol. 62, no. June, pp. 1–5, 2008.

[5] A. Munshi, "OpenCL 1.2 Specification", 2012.

[6] Z. Bozkus and B. B. Fraguela, "A portable high-productivity approach to program heterogeneous systems," *Proc. 2012 IEEE 26th Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2012*, pp. 163–173, 2012.

[7] M. Viñas, Z. Bozkus, and B. B. Fraguela, "Exploiting heterogeneous parallelism with the Heterogeneous Programming Library," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1627–1638, 2013.

[8] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, A. J. Olson, and M. E. T. Al, "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function," *J. Comput. Chem.*, vol. 19, pp. 1639–1662, 1998.

[9] R. Huey, G. M. Morris, A. J. Olson, and D. S. Goodsell, "A semiempirical free energy force field with charge-based desolvation.," *J. Comput. Chem.*, vol. 28, pp. 1145–1152, 2007.

[10] K. M. Micevski D, "Optimizing Autodock with CUDA.," *VPAC Case Study*, 2009. [Online]. Available: http://www.vpac.org/?q=node/290. [Accessed: 01-Jan-2012].

[11] I. Pechan and B. Fehér, "Hardware Accelerated Molecular Docking: A Survey," 2012.

[12] S. Kannan and R. Ganji, "Porting Autodock to CUDA," *Evol. Comput. (CEC), 2010 IEEE Congr.*, 2010.

[13] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew, and A. J. Olson, "Automated Docking Using a Lamarckian Genetic Algorithm and an Empirical Binding Free Energy Function," *J. Comput. Chem.*, vol. 19, pp. 1639–1662, 1998.

[14] I. Pechan and B. Feher, "Molecular Docking on FPGA and GPU Platforms," *Audio Trans. IRE Prof. Gr.*, pp. 474–477, 2011.

[15] I. Pechan, B. Fehér, and A. Bérces, "FPGA-based acceleration of the AutoDock molecular docking software," *Ph. D. Res.*, 2010.

[16] T. VanCourt, Y. Gu, V. Mundada, and M. Herbordt, "Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws," *EURASIP J. Adv. Signal Process.*, vol. 2006, pp. 1–11, 2006.

[17] X. Zhang, S. E. Wong, and F. C. Lightstone, “Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines.,” *J. Comput. Chem.*, vol. 34, no. 11, pp. 915–27, Apr. 2013.

[18] O. Trott and A. J. Olson, “AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading.,” *J. Comput. Chem.*, vol. 31, no. 2, pp. 455–61, Jan. 2010.