N°d'ordre: .....

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University DJILLALI Liabès of Sidi Bel-Abbès
Exact Sciences Faculty
Computer Science Department
EEDIS Laboratory

Doctorate Thesis
3rd *Cycle*

Domain : Mathematics and Computer science
Field : Computer science
Specialty : Computer Science: Information and Knowledge system

By

Mr Hocine SAADI

Metaheuristics on GPU graphics processor; Application to the Molecular Docking problem

Defense on 27-05-2021 in front of the jury :

Pr.	Boukli Hacène Sofiane	Université Djillali Lyabès, SBA	Jury chairman
Pr.	Adjoudj Réda	Université Djillali Lyabès, SBA	Jury member
Pr.	Malki Mimoun	ESI-SBA	Jury member
DR.	NOUALI-Taboudjemat Nadia	CERIST, Alger	Thesis supervisor
Pr.	RAHMOUN Abdelatif	ESI-SBA	Thesis co-supervisor

Academic Year : 2019 - 2020

I dedicate this modest work to my family. I owe everything to my family who encouraged and helped me at every stage of my personal and academic life and longed to see this achievement come true. I dedicate this worrk to my loving mother, generous father, and my honey wife and daughters. Every breath of my life and drop of blood in my body is dedicated to my family. I love you all.

Acknowledgements

First of all, I praise Allah Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study successfully.

Now, I would like to thank gratefully my thesis advisors, Pr NOUALI-Taboudjemat Nadia and Pr RAHMOUN Abdelatif for their availability and continuous support during my thesis preparation. Many thanks to Pr Nouali-Taboudjemat Nadia for her valuable remarks and advises, guidance and encouragement, Many thanks also to Pr. RAHMOUN Abdelatif for his valuable guidance and suggestions. Indeed, their quick feedback and constructive comments.

I would like to thank gratefully Dr Malika Mehdi and Dr Ahcene Bendjoudi for the research theme proposal for this thesis. Many thanks to Dr Malika Mehdi for having trusted me and continues to work with me until the end of this thesis.

I would like to thank the jury members: Pr Boukli Hacène Sofiane and Pr Adjoudj Réda from the Université Djiali Lyabès SBA, Pr Malki Mimoun and and Pr RAHMOUN Abdelatif from ESI-SBA school, and Pr Nadia NOUALI-Taboudjemat from CERIST who honor me by their presence as examining committee in my thesis defense.

I am also pleased to thank Dr Djamal Belazzougui for his help and effort in improving this thesis, his availability to all my requests. Without forgetting to acknowledge my dearest colleagues in DTISI team, Adel Dabah, Nadir Bouchama, Abderezak Seba, Samir Hadjer, and Abdelouahab Amira for the whole time we spent together working, debating, sharing ideas and more.

I would like to thank all members of BIO-HPC Research Group team and especially to José M. Cecilia, Horacio Emilio Pérez Sánchez, and Baldomero Imbernon, with whom I exchanged valuable ideas during my visit to Murcia University, I would like to thank their help to move forward in my research work.

Contents

List of Figures

List of Tables

Introduction...

0.1 Introduction to the context

Several problems from academic, economic and industrial fields can be modelled as combinatorial optimization problems. Most of these problems are complex and often recognized to be NP-hard, making the exhaustive exploration of all solutions impractical because of the combinatorial explosion of the solutions.

This is the case of Molecular Docking(MD) which is the problem of the subject treated in this thesis. MD is a technique used in the pharmaceutical industry in the process of discovering new drugs. Its aim is to predict the binding pose between a small molecule (drug candidate called ligand) and a protein target (the origin of a disease). It consists in calculating the optimal orientation and position of the ligand when bound with the protein to form a stable molecular complex. The objective is to find the best solution (configuration) among a very large set of feasible solutions (the so called search space). Since there is an infinite number of possible configuration, this problem is complex, and known to be NP-hard and CPU time-intensive; its sequential exact resolution in a reasonable amount

of time is impossible. To cope with this difficulty, the problem is modeled as an optimization problem.

Metaheuristics have proven their effectiveness in solving this type of problems as they lead to acceptable solutions within a reasonable time. Several metaheuristics have been proposed to solve the MD problem, such as Genetic Algorithm (GA), simulation annealing (SA), and Particle Swarm Optimization (PSO), etc. Such techniques significantly reduce the computation time needed to examine the entire search space. Nevertheless, these approximate algorithms remain inefficient when dealing with very large instances, since they are greedy in terms of computation time, making it impossible to run them on a simple machine, and often require the use of High-performance computing (HPC), with large computing power and a huge amount of computing resources. Nowadays, such a power can be provided by the Graphics Processing Unit (GPUs) through the use of the parallel computing and the GPGPU paradigm (General-purpose computing on GPUs). The GPUs have lower cost and energy consumption compared with the other HPC solution such as supercomputers.

0.2 The subject of the thesis

The objective of this thesis is to propose, design and implement parallel metaheuristics on GPUs to solve the MD problem with the support of MD related challenges in addition to the new challenges related to the implementation of metaheuristics on modern GPU architectures.

The parallel models of metaheuristics need adaptation and modification to leverage the new requirements of the modern GPU architectures with many issues that have to be taken into account. The latter are mainly related to the latency of the GPU memories and their hierarchical management, divergence of GPU threads and their synchronization, and the optimization of data transferring between the CPU and GPU.

A deep study has to be performed on both types of metaheuristics, population-based metaheuristics (P-metaheuristics) or solution-based metaheuristics (S-metaheuristics), and their data structures, since their data sizes and the access frequencies are different. On the other hand, these data have to be mapped to different kinds of GPU memories with different size and access latency. Thus, during the execution of metaheuristics on GPU, the different threads may access multiple data structures from multiple memory spaces. A key challenge is to efficiently map the threads to the corresponding metaheuristic, for example one neighbor per thread for S-metaheuristic, or a single population per threads block for P-metaheruistic, etc. The coalesced access [1] to the memory is another important issue to deal with to optimize the performance of a GPU-based metaheuristic. The data transferring between GPU and CPU, is another important factor, for which good strategies should be adopted. In addition to the memory access and latency issues, the encoding step is very important when designing a metaheuristic for a specific problem. Thus, the data

[1] A group of threads access to consecutive addresses of the global memory at the same time.

types should be chosen carefully, the ranges of data values should also be analyzed to minimize the memory space required.

Since GPU architecture is based on massively parallel multi-threading, thread synchronization and synchronization requirements of the implemented metaheuristic are important issues that should be dealt about. An efficient parallelization on GPU can be achieved when running a large number of threads, but the order in which these threads are executed is not known. To cope with this challenging issue the programmer has to explicitly manage the threads by the insertion of barrier synchronizations or by using other methods to avoid concurrent accesses to data structures. Metaheuristics contain irregular loops and conditional instructions which can make them hard to parallelize. In addition, threads of the same warp [2] have to simultaneously execute instructions directing to different branches, considering that in a SIMD model threads of the same warp execute the same instruction at a time. Therefore, the different branches of a conditional instruction which is data-dependent may force a serial execution of the different parallel threads of the same warp, considerably degrading the performance of the parallel metaheuristic (Cecilia *et al.* 2013) (Delévacq *et al.* 2013).

To achieve the best performance, an efficient decomposition of the code and a good distribution of tasks between the CPU and the GPU should be adopted. The objective is to leverage the GPU computing power without losing performance in CPU/GPU communication and memory accesses. Thus, the code should be carefully analyzed in order to decide which part of it will stay and execute on CPU, and which compute-intensive part will be sent to the GPU for parallel execution. CPU/GPU communication is another important factor to take into consideration. This communication is done through the global memory which is a relatively slow memory, making the memory transfer between the CPU and GPU time-consuming, possibly inducing a significant degradation of the performance of the application. Therefore, the programmer should optimize the code to minimize the amount of data to be transferred between the host (CPU) and the device (GPU)(Mehdi *et al.* 2013).

The exploration of the huge search space by the search algorithm and the evaluation of the results of the search step by the scoring function are the two big challenges that need to to be addressed in MD. In the last two decades, a lot of effort went to the development of the search algorithms and scoring functions with the aim of coping with these challenges. Despite these huge efforts, some parts of the scoring function remain challenging. For example, the solvation energy calculation which is the most important part of the scoring function is still a challenging part.

Rapidity (speed) and efficiency are the two crucial characteristics of a scoring function in addition to its accuracy (by how much the function fits with the physical reality). For this reason, researchers were forced to develop a simple and acceptable accurate scoring function. Now, with the fast development in computing power (HPC), the coming focus for the scoring function development is how to enhance performance and accuracy. Because of the complexity of the problem the limitation of the computing power in the past, only ligand flexible docking has been addressed,

[2] A warp is a set of 32 threads

protein flexibility is still considered as a big challenge, since higher flexibility needs more computational time to simulate the docking interactions.

0.3 Contributions

We now list the three main contributions of our thesis.

First, we propose a new metaheuristic based on bees swarm optimization (BSO) to model MD as an alternative to the traditional metaheuristics such as Genetic Algorithm (GA), Simulated Annealing (SA), and Particle Swarm Optimization (PSO). BSO simulates the collective honey bee behavior in nature when looking for nectar of the easiest and richest sources using the bees dance protocol. In this algorithm an artificial bee named InitBee works out to find a first solution named Sref with some good features. It then uses Sref as a starting point to find a group of disjoint solutions called space-of-regions to maximally exploit the search space. Subsequently, every bee takes one solution from the space-of-regions group and considers it as its starting point to do a local search (intensification) to look for other potential solutions. To design and adapt our BSO metaheuristic to the molecular docking problem we have to define the components of BSO algorithm: the representation of the problem (encoding solution), the fitness function that evaluates the found solutions, the Search strategy to find disjoint solutions (diversification of solutions), and the neighborhood search performed by each artificial bee (intensification of solutions).

In the second contribution, we design and implement a parallel strategy for the BSO metaheuristic. We propose for that purpose to use the MapCG framework which is based on the MapReduce programming model. The latter facilitates the task for massively parallel and distributed programming. Our aim is to propose a portable application that is independent from the hardware architecture and which can run on CPUs, on GPUs, or on both CPU and GPUs without changing any line of the code. In addition, we propose an efficient calculation of the scoring function on the GPU by evaluating several solutions in parallel. The main idea is to divide the dance table which contains solutions into several pieces and execute them on GPU, then choose the best solution within the set of solutions found in the first step by the reduce function.

In the third contribution, we propose a solution to improve the performance of the blind docking problem (BD). In the last few years, blind docking was proposed to examine the entire surface of target proteins in order to find new interesting binding sites where small molecules (Ligands) can eventually connect to make a stable complex. This method helps to improve the quality of the docking process, but it exponentially increases the computational time. By profiling the docking process, it was noted that most of the CPU time in the docking process is spent in the scoring function. Moreover, for blind docking this time is to be multiplied by the number of binding sites. It is thus of great interest to parallelize this calculation in order to speed up the metaheuristic and the whole docking process. Indeed we propose a new approach to accelerate BD through the exploitation of the emerging GPU architectures with the CUDA programming model and the hyper-Q technique (Bradley 2012). The latter enables

multiple CPU threads to simultaneously launch multiple GPU Kernels [3] to perform calculation. We compared the performances of our parallel approach with the sequential version executed on a single-core CPU and with the parallel multi-core CPU version (48 CPU threads). Our parallel approach was applied to dock a set of protein-ligand complexes from a known benchmark. The experiments show that the GPU-based parallel version outperforms both the multi-core CPU and sequential versions. Indeed, for 100 binding sites, our results show an average speedup of 200x compared to the serial implementation, and 10x compared to the best multi-core CPU version(48 CPU thread). In addition, our parallel approach outperforms the best solution of the literature for solvation energy calculation.

0.4 Thesis plan

This thesis is organized into five chapters:

Chapter one sets out the problematic of the thesis. First, an introduction to the molecular docking (MD) problem is given, followed by a deep study of this field intended to allow the reader to clearly understand the relevant research issues that are addressed in this thesis, like the scoring function, the search algorithm with metaheuristics, and the need for the accelerating of these metaheuristics to enhance the speed and the quality of the solutions to the MD problem.

Chapter two introduces the basic concepts relative to parallelism in a general way. It first presents the different classification of parallel architectures that exist in the literature and the metrics used to evaluate parallel algorithms such as speedup and efficiency. After that, it details the GPUs architecture and show how scientists can leverage this recent architecture to speed up their algorithms. Then, it presents parallel metaheuristics and the challenge faced when trying to implement them on GPU architecture. The second part of this chapter presents a detailed review of the parallel metaheuristics used for solving MD problem on GPU architectures.

Chapter three represents our first contribution in this thesis. It is structured as follows. The background section describes biological and artificial bee swarms and introduces the most well known artificial bees swarm metaheuristics, with a focus on BSO. Section two presents our strategy to solve the docking problem with the BSO algorithm called BSO-DOCK (BSO for Molecular Docking), giving detailed description of all its operators. The last section describes the used dataset and presents some results and discussions.

Chapter four, first introduces the MapReduce model we chose as the parallel model (Dean & Ghemawat 2010). It then introduces the MapCG framework used to implement this model, after that it introduces our GPU parallelization strategies (Liu *et al.* 2017). Finally it gives details on experiments and discusses the obtained results.

The last chapter of this thesis is structured as follows. In the first section, we present the background needed to understand the remaining sec-

[3] Functions in C++/C program when called are executed in parallel on GPU.

tions, such as the Solvation energy term calculation and NVIDIA Pascal GPU architecture (Saadi *et al.* 2019b). Then, it introduces our multi-core CPU and GPU parallelization strategies. It then gives details of the experiments and discusses the obtained results. Finally, last section summarizes the results and presents some concluding remarks.

Finally, the conclusion of this thesis summarizes the major achievement and gives the general appreciation of the results obtained and some perspectives of our work.

0.5 Publications

This thesis gave rise to various written works:

- Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernon, Horacio Emilio Pérez Sánchez, José M. Cecilia.(2020). Efficient GPU-based parallelization of solvation calculationfor the blind docking problem. J.Supercomput.76(3):1980-1998

- Hocine Saadi, Nadia Nouali-Taboudjemat, Malika Mehdi, OusmerSabrine. (2019). A GPU-MapCG based parallelization of BSO metaheuristic for Molecular Docking problem. The International Conference on Parallel and Distributed Processing Techniques Applications PDPTA'19:205-210

- Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernon, Horacio Pérez Sánchez, José M. Cecilia. (2017). Parallel Desolvation Energy Term Calculation for Blind Docking on GPU Architectures. 46th International Conference on Parallel Processing Workshops. ICPP Workshops:16-22

- H.Saadi, Y.Djenouri, N.Nouali, A.Rahmoun, M.Mehdi, A.Bendjoudi. (2016). Bees Swarm Optimization for Molecular Docking). The 6th International Conference on Metaheuristics and Nature Inspired-Computing META'16

- H.saadi, Y.Djanouri, N.Nouali, A.Rahmoune, M.Mehdi, A. Bendjoudi.(2016). Bees Swarm Optimisation for Molecular Docking (Poster). IWBBIO International Work-Conference on Bioinformatics and Biomedical Engineering.

Molecular Docking 1

In this chapter, an introduction to Molecular docking (MD) problem is given. Studying this field allows us to clearly present the relevant research issues that are addressed in this thesis, like the scoring function, the search algorithm using metaheuristics, and the need to accelerating these metaheuristics to enhance the performance and the quality of the MD problem.

1.1 What is Molecular Docking

The study of biological molecules interactions in laboratory is difficult, long, and very expensive. Molecular Docking (MD) is introduced to predict the interaction between two molecules by simulations based on their three-dimensional (3D) structure. The problem is like solving three-dimensional puzzles. It predicts the preferred position, orientation, and conformation of one molecule to a second when bound to each other to form a stable complex (Morris & Lim-Wilby 2008). This method is used in modern structure-based drug design for therapeutic aims (structure based drug design means that we use the 3D structure of molecules to design new drugs with the help of software). MD reduces the time and the price needed to discover novel drugs.

Molecule docking is used also to:

- Study how a molecule is placed relative to another molecule, and visualize the 3D structure of the two molecules before and after interaction.
- Understand how the interaction takes place, which atoms are involved in interaction process, and then answers some biological questions.
- Predict on a large data base of 3D molecules some ones which present the best features and affinities for later laboratory experiments.

Molecule docking method can then be used not only to predict possible positions but also to predict how strong the association between the molecules can be. This is called binding affinity. Indeed, it is useful to know the binding strength when we compare (rank) a group of compounds to determine which one is the best candidate (how strong a compound will bind to the target). By using molecular docking we can save a lot of time and money as use simulation before going to the laboratory to make chemical and biological experiments. In addition, we can predict how a molecule interacts or react. For example, the way towards finding new and safe medications against complex sicknesses needs a huge time of research, from 10 to 15 years with immense finance, sometimes more than one Billion dollars per drug. Grand companies which produce drugs use these methods in the discovery and the development of new medicine.

This method is also very useful when we want to screen ("virtual screening") a number of small compounds called ligands from a synthetic or natural product, to even see if small molecules (ligands) have certain pharmacological effects (inhibitor) on a particular protein that has a bad effect in our body. This inhibitor bind to that protein and deactivate its harmful function (Morris & Lim-Wilby 2008).

All docking methods require a **search method** to explore the search space and to find position and orientation of the ligand, and a **scoring function** to rank the various positions and orientations found (Sousa *et al.* 2006).

In MD the scoring functions are methods to evaluate docking poses, these methods can be empirical, force-field based, or knowledge based.

The search methods fall into two major categories: systematic or stochastic. Systematic search methods sample the search space at predefined intervals, and are deterministic. Stochastic search methods iteratively make random changes to the state variables until a user-defined termination criterion is met, so the outcome of the search varies.

Search methods can also be classified by how broadly they explore the search space, as either local or global. Local search methods tend to find the nearest or local minimum solution to the current conformation, whereas global methods search for the best or global minimum solution within the defined search space. Hybrid global–local search methods have been shown to perform even better than global methods alone, being more efficient and able to find lower energies (Morris *et al.* 1998b).

1.2 Useful definitions for MD

In this section, we give some definitions necessary for understanding the content of this thesis (Chaudhary & Mishra 2016).

- **Molecular modeling:** a general term used to describe the use of computers to design molecules and realize a variety of calculation on these molecules in order to predict their chemical characteristics and behavior.
- **Computational chemistry:** any use of computer to study chemical systems.
- **Receptor :** called also target or host is the receiving molecule, most commonly a protein as Fig 1.1 shows, or another macro-molecule.
- **Ligand:** a small molecule in size compared to the receptor (see Fig 1.2), usually named guest or key. A Ligand binds to the receptor to form a stable complex.
- **Binding site:** The hole in the surface of the protein where the ligand can take place to form a complex with the receptor (see Fig 1.3).
- **Binding mode:** The position, and the orientation of the ligand relative to the receptor, and also the conformation of both two molecules when bound to each other.
- **Pose:** one possible binding mode named also configuration.
- **Search algorithms (Search methods):** the methods to predict the docking poses.

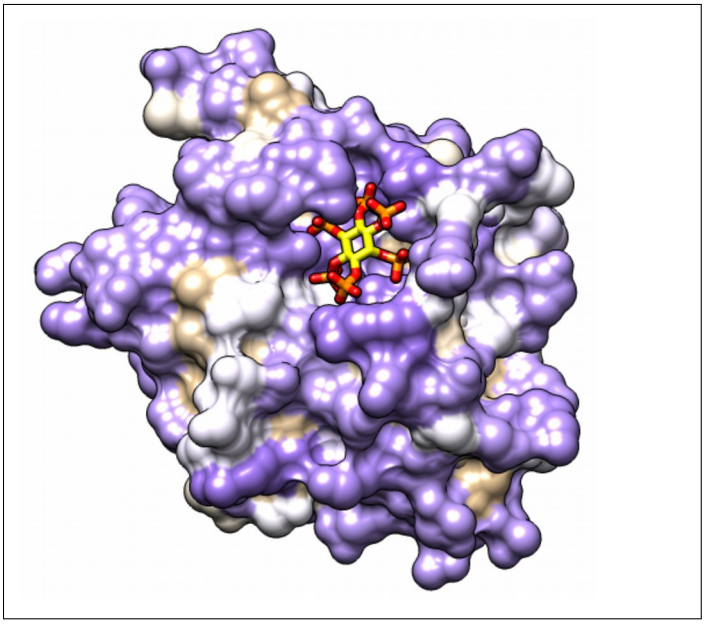

Figure 1.1 – *Receptor Protein surface*

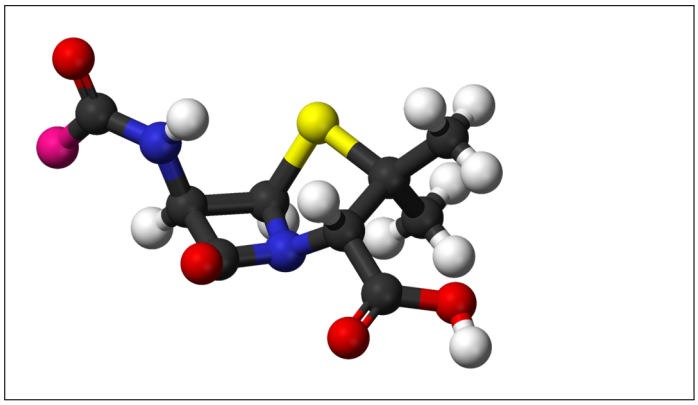

Figure 1.2 – *Ligand represented in 3D structure of atoms*

- **Scoring function (Fitness function):** the methods to evaluate docking poses.
- **Scoring:** the process of evaluating a particular pose by calculation the energy of interaction
- **Ranking:** classifying ligands and find the best ones which can most likely bind favorably to a particular receptor based on the scoring process.

1.3 Docking types

There are two main types of docking in practice depending on the size and the type of the molecules used (Chaudhary & Mishra 2016):

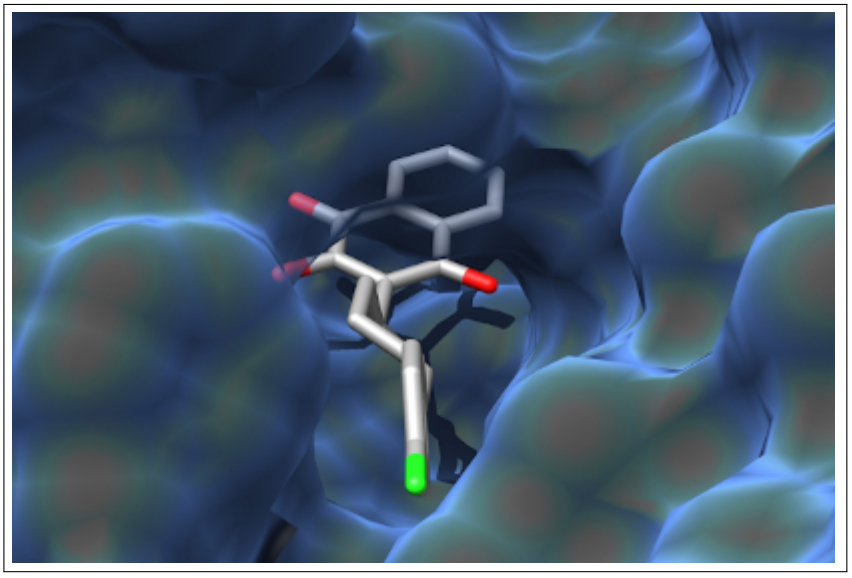

Figure 1.3 – *Small Ligand docked to in a Protein's binding site*

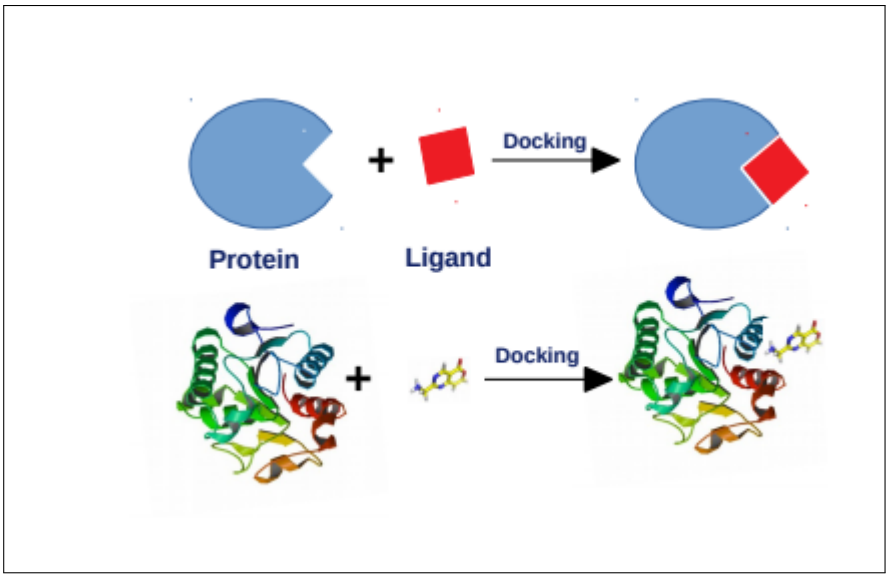

Figure 1.4 – *Ligand-Protein Docking*

Ligand–Protein docking

A ligand is a small molecule compared to the protein which is a macro-molecule, Molecular docking can be thought of as a problem of "lock-and-key", where we are interested in finding the correct relative orientation of the "key" which will open up the "lock" (see figure 1.4). The key is the ligand and the lock is the protein. If the protein is flexible, a Hand–in-glove analogy is more appropriate than lock and key.

Protein – Protein docking

This docking involves two macro-molecules (proteins) which interact with each other by simulation using a computer software. This docking uses the same principle as the protein-ligand docking. However, the complexity increases dramatically. For this reason, most of the solutions proposed in

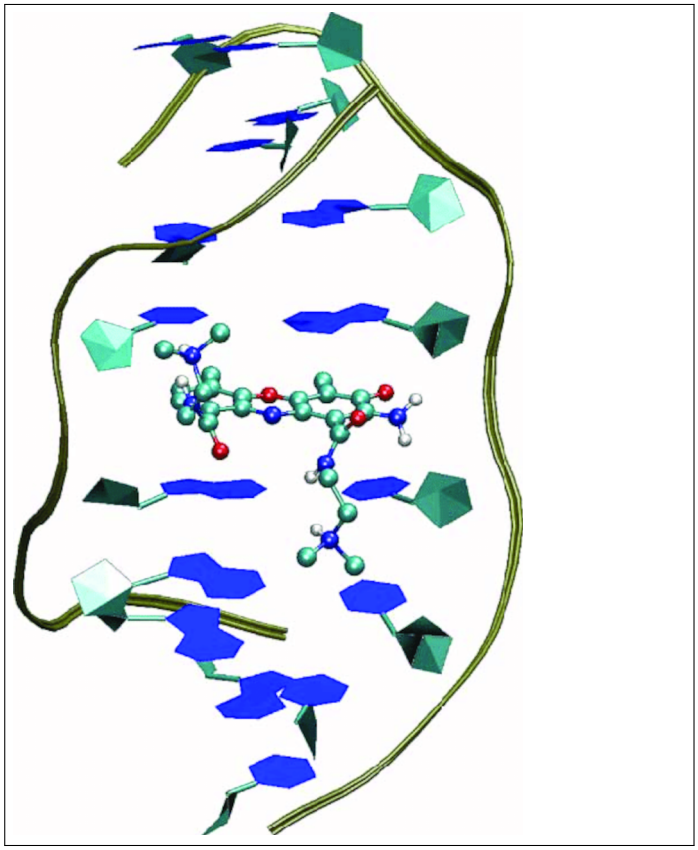

Figure 1.5 – *Ligand-DNA docking*

this type of docking consider the two proteins as rigid molecules (do not change their form). This method provides beneficial information on how certain mechanisms take place in the protein-protein interactions

- Fig 1.5 shows a new kind of docking categorized as Ligand–DNA/RNA docking, or Ligand–nucleic acid (Yan *et al.* 2017). Few researches are done in this field. This docking type is used to study and predict the binding properties between drugs and DNA, for example to find a drug for cancer disease.

If we take into account the complexity as a characteristic of classification, we can distinguish three types of docking (Morris *et al.* 1998a):

Rigid docking

The receptor and ligand are rigid (the structure does not change), the conformation of both is fixed (see Fig 1.6). This method has been widely utilized in docking solutions because it is rapid and does not require a lot of computing power to explore the search space (Halperin *et al.* 2002).

Semi flexible docking

The ligand is flexible (the bonds of the ligand can rotate), the receptor is held fixed, or vice versa.

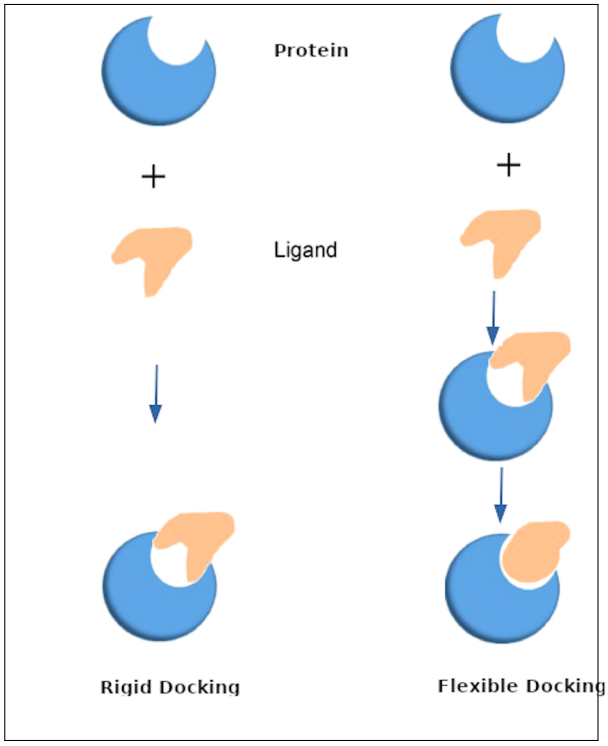

Figure 1.6 – *Flexible and Rigid Docking*

Flexible docking

Both molecules are considered flexible (see Fig. 1.6), it is a computationally demanding method but it gives better results since it reflects what happens in reality (Camacho & Vajda 2002).

The decision of using rigid or flexible docking depends essentially on the calculation resource available, the size and the characteristics of the protein, and the number of ligands if we are dealing with virtual screening.

1.4 Docking approaches

In MD, two approaches are the most used and popular for docking (Chaudhary & Mishra 2016). The first approach uses a surface complementary as criteria to match the protein and the ligand, when the second approach utilizes the energy calculation and simulation of interaction to evaluate the stability of the two compounds. the advantages and limitations of each method are described below.

Shape complementary docking

This approach matches the two molecules by their geometry (see Fig.1.7), it describes the receptor and the ligand as a set of characteristics of their surfaces that make them dockable. These features are the solvent-accessible surface area (SASA) descriptor for the receptor, and the matching surface descriptor of the ligand. Another matching technique is to use the hydrophobic features of the protein. The shape complimentary ap-

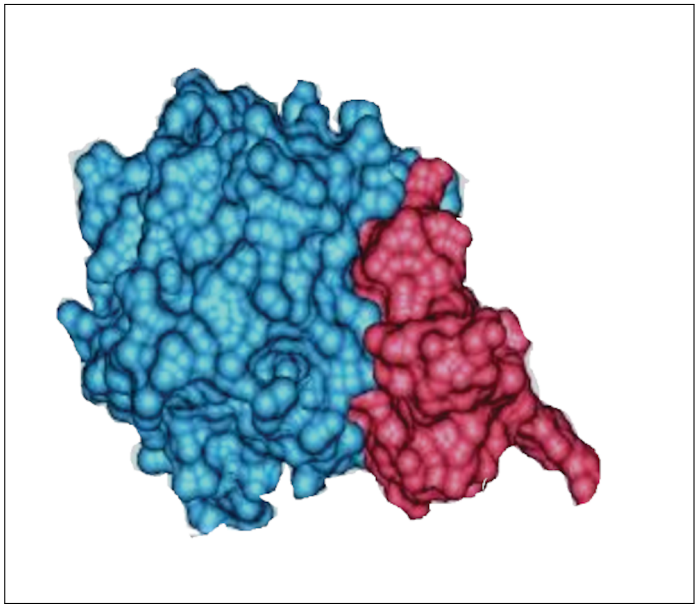

Figure 1.7 – *Docking by shape complementary*

proach is generally fast because it does not take into consideration the conformations of the ligand. It is also scalable to protein-protein interactions. However, it does not take into consideration all the types of energy interaction between the two molecules.

Energy based docking

This approach simulates the atoms interaction between the two compounds (see Fig.1.8). It is the most difficult docking method. It simulates the moves of the ligand to find a good position in the active site of the target protein. The moves include translations of the center of the ligand and its rotations in the binding site. It also include its conformation which is the structural changes of the molecule defined by the torsion angle rotations. Then the energy of each of these moves is calculated to evaluate it.

The clear advantage of this approach is the ability to model the ligand flexibility whereas shape complementary approach has to use some intelligent techniques to model the flexibility of ligand or protein side. In addition, this approach is substantially closer to what happens in protein-ligand interaction in reality. An evident disadvantage is that this approach is time-consuming which needs huge computation resource to calculate the energy of all poses found and rank them to find the optimal one.

1.5 Docking Challenges

The **scoring function** and **the search algorithm** are the major two challenges to face in molecular docking

In the last two decades, lot of effort has gone into developing the search algorithm and the scoring function, this effort has almost settled the search algorithm problems. Although some parts of the scoring function remain challenging. For example, the solvation energy calculation persist

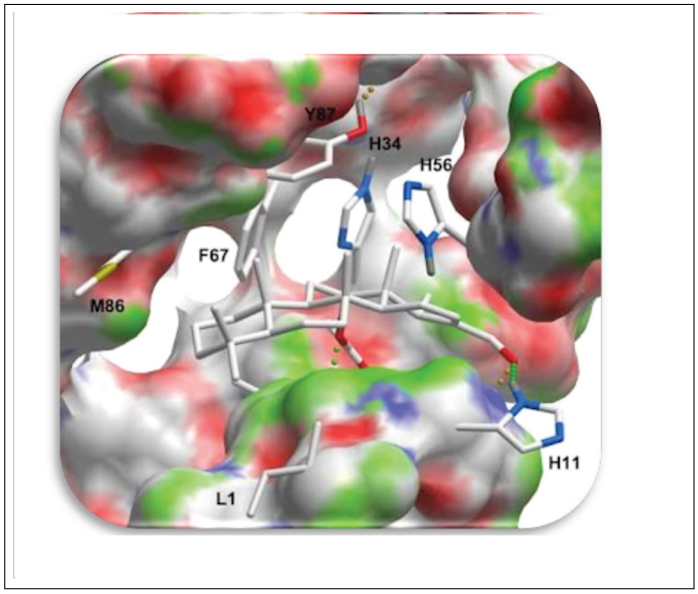

Figure 1.8 – *Docking with simulation approach*

the main challenging part, in particular, for the force field scoring functions (Dar & Mir 2017).

Rapidity (speed) and efficiency are the two crucial characteristics of a scoring function. In the past, researcher were forced to develop a simple and acceptable scoring function because of the calculation power limitation. Now with the fast development in computing calculation power, the coming focus for the scoring function development is how to enhance performance and accuracy (Huang & Zou 2010).

Because of the complexity of the problem which increases with the degree of freedom (ligand's bonds which can moves), and the limitation of the computing power in the past, only ligand flexible docking with protein chain side flexibility has been addressed recently, protein flexibility is considered as a big challenge where higher flexibility needs more computational time to simulate the interactions. The development of docking methods with protein flexibility still in its beginning and thereby stands one of the leading future research directions in docking between protein and ligand.

1.6 The Search Algorithm

The search algorithm is used in MD to explore the search space to find the best ligand poses among millions of ligand conformations investigated. We can distinct two major classes of searching algorithms:

Systematic Search algorithm

Systematic search algorithms are used to generate all possible ligand binding poses by exploring all the search space of the ligand when bounded to the protein. There are two major categories of those searching methods: exhaustive search and fragmentation search (Huang & Zou 2010).

The simplest systematic methods are exhaustive search methods which

use exact search algorithms. In these methods, docking is performed by looking for all possible conformations. However, the number of the solutions becomes gigantic (combinatorial explosion) with the increase of the degree of freedom. Thus, this method can be used for rapid virtual screening with rigid docking, after that the best ligands are docked with more accurate search methods (Huang & Zou 2010).

In the second systematic search called fragmentation methods; the small molecule is split into several fragments. Then, the conformation of the ligand is built adding fragments incrementally in the binding site and linking them. DOCK (Ewing *et al.* 2001), and FlexX (Pagadala *et al.* 2017) are the most known tools that use fragmentation methods.

Stochastic Algorithms

In these algorithms, the position, the orientations and the conformations of the ligand are found by making random changes in the search space. It is generally based on meta-heuristics (Huang & Zou 2010). The most used algorithms in this methods are; Evolutionary Algorithms (EA), Swarm Optimization (SO), Tabu Search, and Monte Carlo (MC) methods.

Evolutionary algorithms (EAs) are inspired from the evolutionary process in biological systems, and use this evolution process to look for the best ligand poses. The most known algorithms of EAs are the genetic algorithms (GAs). The popular docking software that implement evolutionary algorithms are: AutoDock, MolDock, GOLD, DIVALI, FLIPDock (Pagadala *et al.* 2017).

Swarm optimization meta-heuristics (SO) model the swarm intelligence to explore the search space and find an optimal solution. The conformation and the position of the ligand are calculated based on the neighbors' information. SODOCK, Tribe-PSO, and PSO@Autodock are examples of popular solutions that use swarm optimization meta-heuristics as search algorithm(Pagadala *et al.* 2017).

Tabu search is a local search method, the acceptance or rejection of a solution depends on the neighborhoods search area explored previously. A docking parameter called RMSD of the current solutions is calculated (Raschka 2014), then it is compared with the RMSD of the previous registered solutions, some probabilities are used to accept or reject this solution. PSI-DOCK, PRO, and LEADS are examples of docking softwares that use tabu-search methods (Pagadala *et al.* 2017).

Monte Carlo method uses the Boltzmann probability (P) as shown in equation 1.1, this P is calculated for each conformation and used to accept or reject this conformation

$$P \approx e^{[-(E_1 - E_0) \div k_B * T]} \tag{1.1}$$

In equation 1.1 E_0 is the initial energy score (the energy of interaction between molecules which represent the scoring function) before conformation change, E_1 is the energy scores after the conformation random change, k_B is the Boltzmann constant, T is the absolute temperature of the

system. ICM, MCDOCK, QXP, Prodock, are example of docking software that use Monte Carlo method (Liu & Wang 1999).

1.7 The scoring Function

The scoring function is a set of mathematical functions which evaluate the results of the search algorithm (Huang & Zou 2010). This function is used to approximately evaluate the protein-ligand interactions, and predict the binding affinity and the strength between the two molecules after the docking process. Below, we give the mathematical formulation of this function. A good scoring function with a good quality is computationally too expensive. To reduce the complexity, more simple terms must be used while keeping an acceptable cost in term of accuracy. A compromise between the speed of calculation and the accuracy of the scoring function represents a challenge to be met (Sousa *et al.* 2006). There are typically three general classes of scoring functions, force-field-based, empirical, and knowledge-based.

Force Field-Based Scoring Function

This type of function is called also molecular mechanics-based scoring functions (force field energy) (Chaudhary & Mishra 2016). Most of the MD scoring functions belongs to this category, where the binding affinities are calculated by summing the strength of different intramolecular and intermolecular interactions between all atoms of the ligand and the target protein (Liu & Wang 2015). The energy forces used are;

- van der Waals energy
- electrostatic energy
- hydrogyn bonds energy
- solvation (desolvation) energy

The following equation represents the most used force field scoring function

$$G_{binding} = E_{vdw} + E_{electrostatic} + E_{H-bond} + E_{desolvation} \tag{1.2}$$

Empirical Scoring Function

In the empirical scoring functions, the affinity energy is calculated by approximations, using a set of solved docking complex. It is based on reproducing experimental data to determine the coefficients (Wi in equation 1.3) for the different terms of the scoring function (Dar & Mir 2017)(Sousa *et al.* 2006).

$$\Delta G = \sum W_i * \Delta G_i \tag{1.3}$$

In this equation Gi represent individual empirical energy terms such as VDW energy, electrostatic energy, hydrogen bonding energy, desolvation term, entropy term, hydrophobicity term...Etc (Huang & Zou 2010).

The advantage of this method is the easy calculation of the various terms. However, the principal disadvantages of these techniques are their dependency on the experimental data (training set) utilized in the parameterization procedure (Huang & Zou 2010).

Knowledge- Based scoring Function

This method uses a database of known ligand-protein complex structure (Chaudhary & Mishra 2016), it is based on statistical observation on the frequency of occurrence of different atoms pair contacts in the protein-ligand compounds, which contributes favorably to binding affinity (Sousa *et al.* 2006). It deals with the very simple type of interactions between atoms. However, the main disadvantages of this class can be summarized in the reliance on limited available data on known complex structures.

In the last decade, new scoring functions based on machine learning have been developed (Ballester & Mitchell 2010) (Silva *et al.* 2014) (Kinnings *et al.* 2011). The aim is to outperform the traditional classes of scoring functions cited above as the scoring function is induced.

1.8 Docking Workflow

The general process used in the most docking methods involves more or less the following steps (see Fig.1.9) [7 sur les article formart papier]

Step I - Preparation of the protein and the ligand

The protein represented in its 3D structure is downloaded from the Protein data bank (PDB) (DataBank 2017). The ligands can also retrieved from a ligands data base such as ZINC database (Irwin & Shoichet 2017), PubChem database (National Institutes of Health (NIH) 017) or it can be designed from scratch by using some tools like Chemsketch tool. The protein structure should be prepared by removing the water molecules and heterogeneous atoms if present, adding or removing some charges, adding some residues (molecules). If we are dealing with protein flexibility we should generate the flexible side chains of this protein (the atoms of the protein which interact with the ligand). For the ligand, the most important treatment is to fix the degree of freedom by choosing the number of dynamic bonds. If we choose the ligand from a database, the rule of five of LIPINSKY should be applied to increase the chance of the success of the docking (Huang & Zou 2010).

The five rules LIPINSKY are:

1. Less than five hydrogen bond donors.
2. Less than ten hydrogen bond acceptors.
3. Molecular mass less than 500 units.
4. High lipophilicity (expressed as LogP not over 5)
5. Molar reactivity should be between 40-130

If two rules or more are not satisfied, the ligand is non-drung like and should not be used for docking.

Step II – Active site prediction

After the preparation of the protein and the ligand, the active site of the protein should be found by using specific algorithms. The protein could contain many active sites; the best one should be chosen (Halperin *et al.* 2002) (Morris *et al.* 2010).

Step III- Docking

After preparing the ligand and the protein, the docking process can take place by using a software such as Autodock or Gold. The scoring function evaluates scores on the basis of poses given by the search algorithm and picks out the best one. The scoring function is also used to rank the best ligands from a database in case of virtual screening.

Step IV Analyzing of the results

Finally, the results of the docking process are analyzed. The best conformation of the ligand is chosen based on the best energy (lowest), and the good stability with the protein target (number of H-bonds formed between ligand and protein).

1.9 Docking softwares

There are many commercial or free open source docking programs available which are able to simulate the protein interaction with the ligand. New algorithms from industry and academia are quickly incorporated into the high end packages. Free open source packages are becoming more stable and offering functionality that rivals some of the commercial software. The success of a docking program mainly depends on two components: the search algorithm and, the scoring function. More than 60 softwares for docking are created during the last 20 years, Here we summarize the most known and used software for docking (Chaudhary & Mishra 2016).

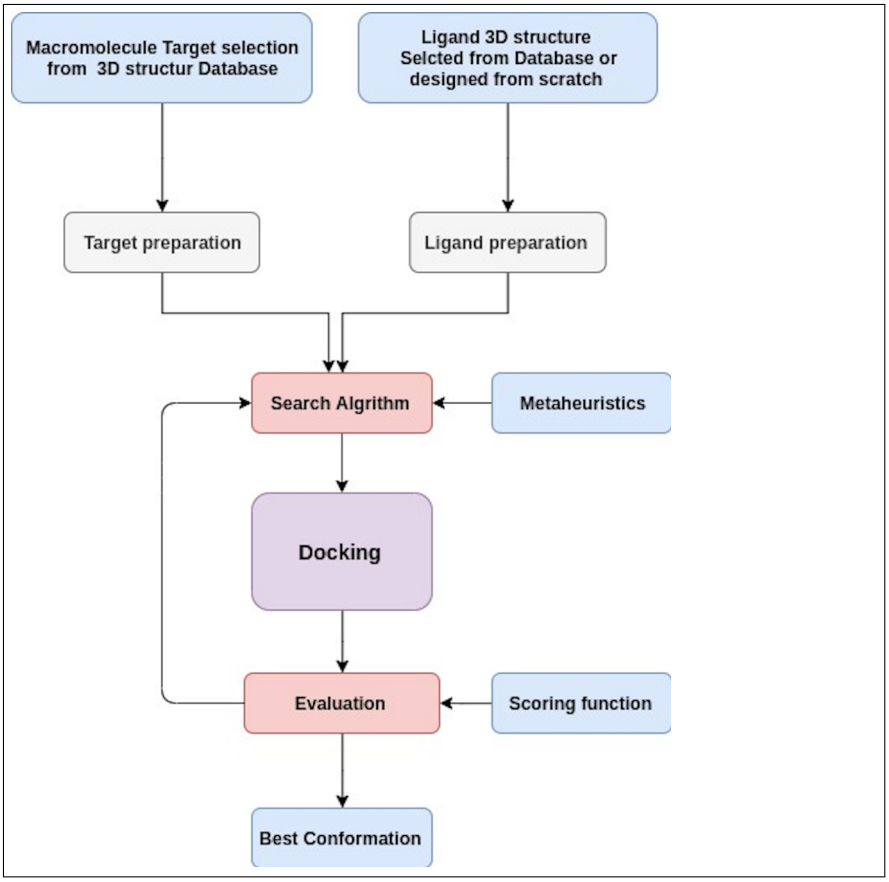

Figure 1.9 – *Docking workflow*

1.9.1 Autodock

Autodock is a popular and one of the most cited docking software in research community, it is open source and free for academic uses. Autodock is the first docking program which performs full flexibility docking between ligand and protein (Morris *et al.* 2010). It combines two methods to achieve docking: rapid grid-based energy pre-calculation and efficient search algorithm (Morris *et al.* 2009). The current version of Autodock primarily uses Lamarckian Genetic Algorithm (LGA) metaheuristic which is a Genetic algorithm GA combined with a local search method (Solis and Wets), The user can also utilize other metaheuristics offered by this software like simulated annealing (SA) or simple genetic algorithm (GA). To evaluate conformations, the Semi-empirical force field free energy scoring function is used. The actual version of Autodock contains two primary programs: autodock used to perform the docking of the ligand to a set of energy grids that describes the target macromolecule and autogrid which pre-calculates these grids.

The docking process with Autodock is performed in several steps:

1. Preparation of coordinate files with PDBQ format for both ligand and protein by using AutoDockTools (ADT). ADT is a set of tools to prepare molecules, visualize and analyze the results.

2. Pre-calculation of the atomic energy using Autogrid program
3. Docking of ligand with the target protein using autodock program
4. Analysis of results using AutoDockTools.

Autodock Vina is an improved version of autodock with an improved local search algorithm and a simple and rapid scoring function compared to Autodock (Trott & Olson 2010). Vina allows the use of multicore CPU. It uses the same input files format as autodock (Pdbqt) and AutodockTools can be used also for visualizing results.

Autodock is a free open source software accessible under the GNU General Public License (GPL) for academic use. Autodock Vina is under the Apache license available for commercial or academic use.

1.9.2 GOLD

Genetic Optimisation for Ligand Docking (GOLD) is a good software for docking flexible ligand into protein binding sites and for virtual screening. The software packages include a software tool called Hermes which is a graphical user interface for docking with Gold. Hermes is used also to prepare input files, and to visualize the docking results (Verdonk *et al.* 2003).

The scoring function of Gold is a molecular mechanics function with 4 terms as defined in (Verdonk *et al.* 2003) and shown in Equation 1.4.

$$S = S_{hb} + S_{vdw} + S_{hb-int} + S_{vdw-int} \tag{1.4}$$

Where:

- Shb: the hydrogen bond energy of the complex protein-ligand
- Svdw : the complex van der Waals energy
- Shb-in: the intramolecular hydrogen bonds energy in the ligand
- Svdw-int : the intramolecular van der Waals energy in the ligand

The search algorithm used is the genetic algorithm (GA) meta-heuristic in which some parameters are modified.

Gold uses a unique method to put the ligand in the binding site. This mechanism is based on a set of fitting points added to the hydrogen-bonding groups on both molecules, then mapping receptor points on the ligand with donor points in the protein and vice versa (Verdonk *et al.* 2003).

Gold has only Commercial software license.

1.9.3 DOCK

The first version of DOCK is one of the oldest docking programs. It uses geometric matching technique with rigid body and fragment based algorithm, which make it a fast docking tool. The force-field based scoring function is used for binding affinity (Ewing *et al.* 2001).

The current version named DOCK 6 is open source software. The scoring function is improved in this version by enhancing the Solvation energy term. Ligand and protein flexibility are allowed. With this version researchers can also use De-novo design docking method (Lang *et al.* 2009).

1.9.4 FlexX

Flexible ligands and grid of the proteins are used in Flex. This software is a fragment based method where conformations are generating by using MIMUMBA torsion angle data base (Morris & Lim-Wilby 2008). FlexX uses a scoring function called Boehm with some adaptations made for docking purpose (Pagadala *et al.* 2017).

FlexE is the newest version, it allows protein side chain flexibility, and therefore it enhances the accuracy of the docking.

The most used and known docking software and their features and uses are surmised in Table 1.1. (Fan *et al.* 2019).

Table 1.1 – *Molecular docking software*

Software name	Search algorithm	Scoring Function	Uses and features
Autodock	Genetic Algorithm GA,local Genetic Algorithm LGA,Simulated Annealing SA, Particle Swarm Optimization PSO	Semi-Empirical molecular force field	General Public License (GPL) Free open source for academic uses Rigid or flexible docking
Autodock vina	GA with simple Local search	Fast Semi-Empirical	Rigid or flexible docking Faster than Autodock Not suitable for ligands charged
Gold	Genetic Algorithm GA	Molecular mechanics	Commercial software license Evaluated as a very accurate software Rigid or flexible docking
Dock	Fragment based Algorithm	Molecular force field	Fast docking software Geometric matching technique Flexible Docking
FlexX	Fragment based Algorithm	Semi-Empirical	Fast docking software Rigid or flexible docking Can be used for Virtual screening
ZDOCK	Molecular dynamics	Molecular force field	Geometric matching Rigid Docking
LeDOCK	Genetic Algorithm GA Simulated Annealing SA	Molecular force field	Fast docking software Recommended for virtual screening

AutoDock and GOLD are the popular and the most cited and top-ranked docking tools. However, they are not necessarily the most accurate. Each of them has many advantages, but also has several limitations (Rawal *et al.* 2019). Therefore, the user is always advised to carefully choose the docking tool based on the details of each program and the size and the nature of the molecules to be docked.

1.10 Molecular Docking tools

In this section we present the most known and used tools which help researchers to perform a good docking,these tool can be used to prepare the molecules for docking, and to visualise the results.

1.10.1 Autodock tools

AutodockTools ADT is a docking user interface tools developed by the same laboratory that develops Autodock. With ADT users can prepare input ligands and proteins files, visualize molecules in 3D and analyse the results of the doking process ran with Autodock software (El-Hachem *et al.* 2017). The users use ADT to prepare the ligand and the protein molecules by adding hydrogen atoms or atomic charges, converting input molecules to a convenient PDBQT format appropriate with Autodock, define rotatable bonds in the ligand (flexibility). Then they can create a grids parameter file GPF and a docking parameters file DPF. And finally visualize the results in 3D images.

1.10.2 Paymol

PyMOL is an open source tool, created by Warren Lyford DeLano and maintained by Schrödinger laboratory [24]. It is a commercial software, but free for academic uses. Initially created for molecules visualization in 3D, however, it can be used for Molecular docking simulation. Many plug-ins are developed to allow the researchers to prepare and visualize MD with PyMol. As an example we can name Autodock and Autodock vina plugins (Seeliger & De Groot 2010).

1.10.3 UCSF Chimera

Chimera is a good software to analyze and visualize molecules interaction, and molecules structures. This software is used also to visualize the docking results. It creates a high quality images and movies (Pettersen *et al.* 2004). It is well documented and free for noncommercial use. Chimera is introduced by the university of California, San Francisco. And it is supported by the National Institutes of Health of USA.

1.11 Conclusion

In this chapter we introduced the MD domain and detailed the two key elements which ensure the success of the MD process which are the search algorithm and the scoring function. MD is an optimization and an NP-hard problem; accordingly, metaheuristics are needed to render the issue tractable with an accepted time span. However, when we deal with huge data-sets, these metaheuristics require acceleration. The solution to this issue is to parallelize those metaheuristics by using the high performance computing (HPC) resources like accelerators. The scoring function is a major challenge in structure based drug design. The efficiency and the accuracy of the docking process depend on the quality and the speed of this scoring function.

In the next chapter, we present the parallel metaheuristics, the architecture of the modern GPUs, and the stat of art of parallel metaheuristic on GPU used to solve the MD problem.

Parallel Metaheuristics on GPU

2

2.1 Introduction

In the following, we introduce the basic information on parallel computing and parallel metaheuristics. A clear understanding of GPU characteristics is required before providing an efficient implementation of parallel metaheuristics on GPU. So we give details about GPU architectures in the first part. In the last part of this chapter, we focus on the existing works on solving the molecular docking problem with parallel metaheuristics on GPU architectures.

2.2 Parallel Computing

Parallel computing is the ability to run multiple tasks or computations at the same time, all nowadays computer systems have parallelism technology implemented at many scales, starting from parallelism in nano-circuits to reach parallel systems that occupy large data center. There are divers reasons for why parallelism has become popular over the past years which are related to the computation time and the energy consumption. First, parallel computing is commonly faster than sequential computing, second efficiency in terms of power consumption, especially when using GPUs (Acar & Blelloch 2019).

Parallel Hardware

In the last two decades multiple processing units (cores) have been placed onto a single chip. These processors can be special purpose, like those found in Graphics Processing Units (GPUs), or general purpose processors. In the last years, most of computing devices such as mobile phones, desktops and servers used multi-core chips and take advantage of this parallel hardware. At a larger scale, many computers can be connected by a network and used together to solve large problems (Acar & Blelloch 2019).

Parallel Software

The way of organizing the computation in parallel programs is very different than the sequential programs. This is the big challenge because the computation in parallel software must be independent to be performed in parallel, and the developer has to identify dependencies in the computation and avoid creating unnecessary ones. Another important challenge is how to implement the parallel algorithms, and which programming language and system to use for coding, knowing that these different programming languages and systems frequently target a particular kind of hardware and are used to solve a particular problem.

2.3 Parallel architectures classification

Numerous classifications of parallel architectures have been proposed in the literature. The best known is the one proposed by Flynn in 1966 (Flynn 1966). This latter is based on instruction flows and the way they are applied to the data flow. A parallel or concurrent operation can have different forms within a computer system based on the different flows used in the calculation process. There are four classes based on the Flynn classification (see Figure 2.1).

Single Instruction Single Data stream (SISD)

SISD is a term for a hardware architecture in which a single instruction is executed by one CPU using a single data as input (Flynn 1995). The execution of a new instruction starts only at the end of the previous one.

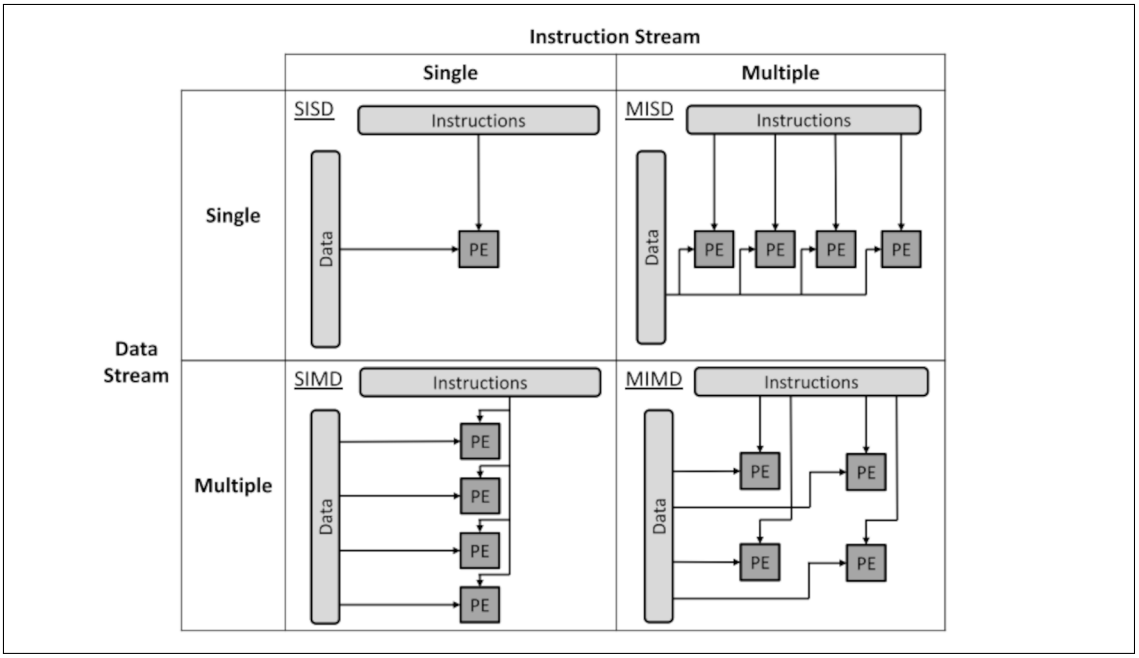

Figure 2.1 – *Flynn's taxonomy*

Therefore, there is no parallelism. The classical machine of Von Neumann is a typical example from this class.

Single Instruction Multiple Data (SIMD)

This class is called the massively parallel architecture. SIMD machines incorporate many processors managed by one or many control units. All units of calculation execute the same instruction or program received from the control unit, but operates on different data. The units of calculation execute the same instruction at each unit of time (operate synchronously). The two most popular categories of SIMD processors are the vector processor and the matrix processor. GPUs and newer CPUs belong to this architecture

Multiple Instruction Single Data (MISD)

The MISD architecture can conceptually be represented by several independent processors operating on a single data stream, transmitting the results from one unit to another one. In terms of micro architecture, this is exactly what the vector processor does. However, in the vector pipeline the operations are simply fragments of a larger operation (Flynn & Rudd 1996). The data processed during each processing step in the pipeline is different from the previous one. According to some authors and experts, this class does not correspond to realistic model of operation (Matloff 2006).

Multiple Instruction Multiple Data (MIMD)

In this class, the processors are independent of each other and work asynchronously. Each processor has its own control unit, and its own data like SIMD model. However, it can also have its own flow of instruction, it can perform its task independently of the others even though most applications require some form of synchronization between processors to exchange information and data with each other (Flynn & Rudd 1996). The MIMD is the most common architecture of parallel processors. Multiprocessors machines belong to this class.

To ensure data coherence, it is often necessary to synchronize the different processors with each other, the synchronization techniques rely on the memory organization.

If we take memory access and uses as a criteria of classification, we can distinguish two major classes; shared memory machines and distributed memory machines

Shared Memory machines

A shared-memory machine is mainly made of processors with independent clocks working asynchronously, and communicating via a single and same memory (the shared memory) by dropping and reading their data using this memory (see Figure 2.2). An additional difficulty is that each processor usually has at least one data cache memory. All of these caches memory must have coherent information at the crucial moments. The Synchronization can be performed by using:

- Synchronization barriers.
- Semaphores with the two operations, P and V.
- Mutexlock by using binary semaphore, used to protect a critical section.
- Monitors: high-level construction, implicit lock.

Distributed Memory machines

Each processor has its own memory and cannot access to the others memories. Information is communicated and exchanged between the processors in the form of messages, either synchronously or asynchronously. Processors are not all connected to each other because of the high cost of these types of connections. A message between two processors that are not immediately neighbors will take a path composed of a succession of processors (see Figure 2.3).

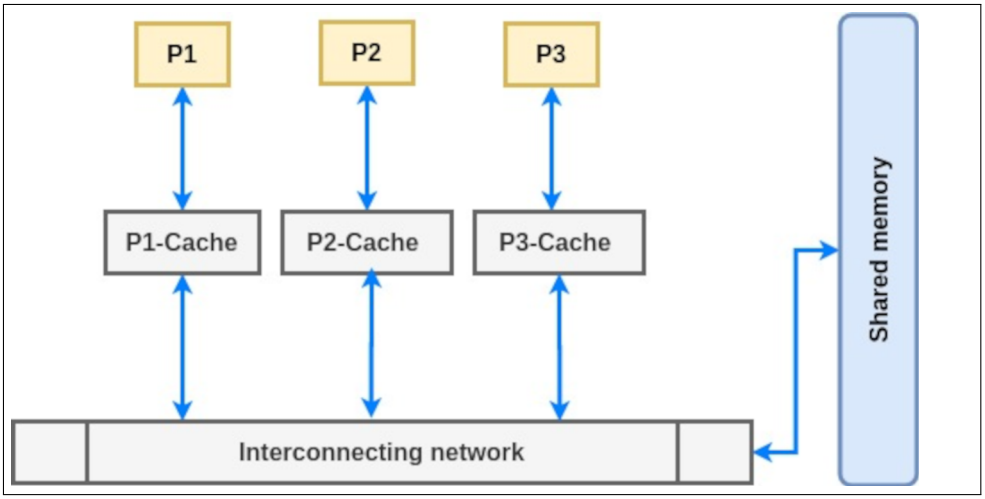

Figure 2.2 – *Shared memory machine*

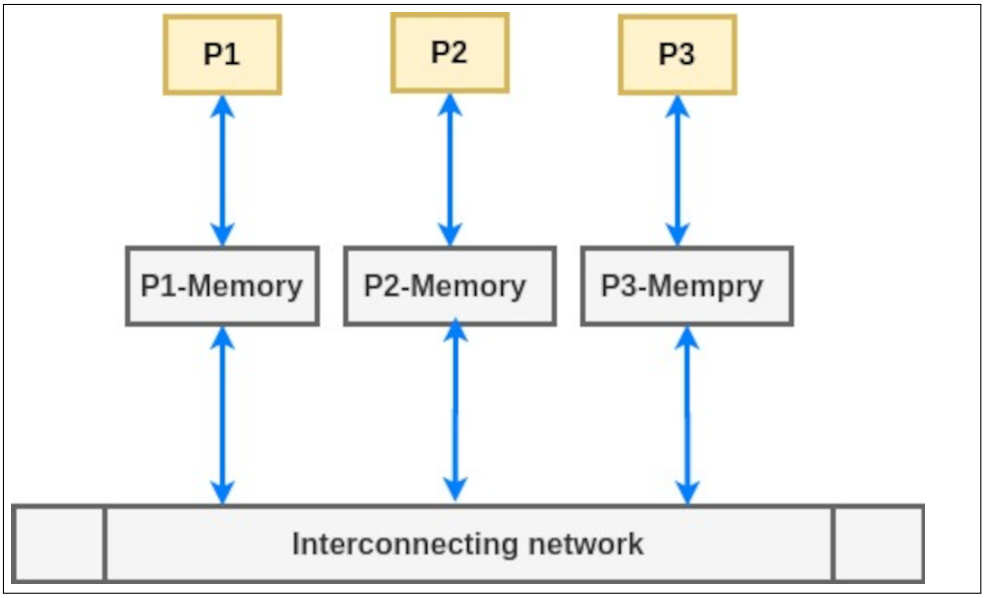

Figure 2.3 – *Distributed memory machine*

2.4 Acceleration and efficiency

Efficiency and acceleration are the most used criteria to evaluate the performance of parallel algorithms.

The efficiency for sequential algorithms is commonly measured by the execution time, the later depends typically on the size of the problem and input data (Cormen *et al.* 1994).

The accelerating or speed-up Sp for parallel algorithms is defined as follows: Let n be the size of the input data, and we use P parallel processors. Sp is the ratio between the time of the best sequential algorithm $Ts(n)$ and the time spend by the parallel version to solve it $Tp(n)$. So, the speed-up Sp of the parallel version that solves the problem X is defined by the Equation 2.1:

$$Spx(n) = Ts(n) \div Tp(n) \tag{2.1}$$

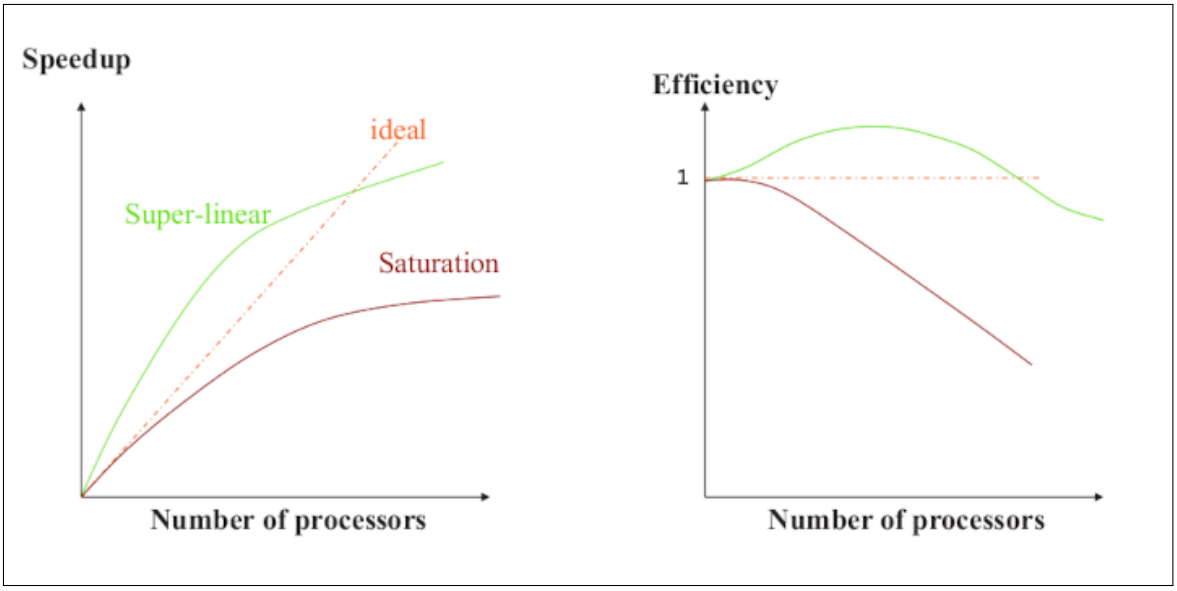

Figure 2.4 – *Different cases of speed-up and efficiency*

The efficiency E represents the second performance measure of a parallel algorithm, it is represented by the following equation:

$$Ex(n) = Ts(n) \div (p * Tp(n)) \tag{2.2}$$

which is equal to :

$$Ex(n) = Spx(n) \div P \tag{2.3}$$

This measurement gives an indication about the efficiency of using P processors in the parallel algorithm. An efficiency value equal to one indicates that the parallel algorithm runs P time faster using P processors as compared with the best serial algorithm with on processor. Thus, each parallel processor accomplishes its tasks efficiency during each step of the parallel algorithm (JéJé 1992). We can notice that efficiency is closely related to the number of processors. In addition to its use in comparing the sequential and parallel algorithm, efficiency can also be used to compare the performance of two parallel algorithm, i.e., the greater the efficiency, the better the parallel solution is. As shown in the Figure 2.4, an ideal parallel speed-up implies an efficiency equal to one. For some kind of algorithm and optimization problems, we can get a super-linear speed-up characterized by an efficiency greater then one. This situation can appear when the parallel algorithm avoid some computation performed in the sequential algorithm. We say that a parallel algorithm is scalable if it remains efficient for a large number of processors.

The speed-up and efficiency depend on the size of the problem and one the number of processor. By fixing the size of the problem, it is pos-

sible to study the evolution of these indicators (speed-up and efficiency) according to the number of processors. The converse is also possible, by fixing the number of processors, we can study the evolution of the acceleration and the efficiency according to the size of the problem. To this aim, the *iso-efficiency* criteria is used. This relationship can be useful for determining the ideal number of processors for a given instance of the problem or to estimate the minimum size that a problem instance should have, in order to get some given speed-up.

2.5 Performance factors of parallel algorithm

To write a good and an efficient parallel algorithm, programmers should take into consideration some important factors which have a big impact on the performance of this parallel algorithm. In the following, we present some of these factors (Codenotti & Leoncini 1992).

2.5.1 Granularity of parallelism

The parallelism grain means the average size of jobs assigned to the processors. The size is generally measured by either the execution time, the used memory, or the number of executed instructions. Two types of granularity of parallelism may be determined; coarse grain parallelism and fine grain parallelism. The choice between these two kinds is firmly related to the characteristics of the used parallel machine. Principally, we start by considering coarse grained parallelism before going to fine grained. Fine grained dose not require a deep knowledge the code contrary to coarse grained parallelism. But, it needs many synchronization point, and several part of code need to be run in parallel. The coarse grained uses a limited number of synchronization which gives a higher parallelization gain. However, it requires a good knowledge of the code.

2.5.2 Communications

An important part of the execution time of a parallel algorithm is devoted to information interchange between the processors which represents the communication load . This later can affect the performance of this parallel algorithm according to the used communicating model. In a shared-memory model, all processors have access to the common memory, the communication load in this model depends on the size of the data exchanged between all processors, the memory access time, and the conflicts management of the simultaneous read and write of the same data. In a distributed memory model, the communication load in this kind depends on the network topology, the network bandwidth (the network capacity of data transferring), the size and the number of messages sent, the structure of the algorithm, and the number of the processors used.

2.5.3 Synchronization

In parallel programming synchronization points are used to organize some order of events that are dependent on each other, in order to guaran-

tee that all the processors have completed their parallel task. In a shared memory model, synchronization can be performed using either barriers or semaphores. If a barrier is used, no processor can continue running after that point until all the other processors have reached it. Semaphores are used to control a critical area where only one processor can execute certain instructions in the algorithm. In a distributed memory model, synchronization is achieved by using blocking send and receive operations. During the synchronization, processors remain inactive for a period of time that varies according to the algorithm. Synchronization can be detrimental to the efficiency of the parallelization, programmer should minimize the synchronization points as much as possible.

2.5.4 Task decomposition

The aim of task decomposition is the distribution of computation and the data across all processors, in order to minimize the inactive time of processors during synchronization and to maximize the efficiency. Calégari (Calégari 1999) proposed three task allocation strategies. These strategies depend on the time of the allocation and on the number of tasks. If they are both determined at the compilation time, then the static allocation strategy is used. Contrarily, if they both vary during the execution, then the adaptive allocation strategy is used. If the number of tasks is determined at compilation time and the allocation is performed during execution, then, a dynamic allocation strategy is used. In the last two strategies, a load balancing algorithm is needed to balance the load between processors. Generally the static allocation is used if the computation load of the different task is approximately the same and the number of tasks is known in advance. However, if the tasks are heterogeneous, then the dynamic strategy will probably be sufficient.

2.6 Current parallel Computer Architectures

In order to build efficient parallel programs, it is important to be aware of the trends in the recent computer architectures. The classification of current parallel computer architectures are summarized as follows (Geshi 2019):

• Multi-core : In this class, the number of cores (computational units), is increased to achieve a good speed-up. The number of cores is around 10 in the basic CPU models of this architecture, it can reach more than 30 cores in the high-end CPU classes.

• Many-Core: The parallel architecture is the same as multi-core architecture. However, the number of cores is greater and the frequency is invented to be lower. The number of cores is about 60 physical cores. From 2018, the parallelism is more than 200.

• GPU (Accelerator): In the last decade, Graphics Processing Units (GPUs) are commonly used as accelerators in addition to CPUs. In the following , We will detailed the GPUs architecture.

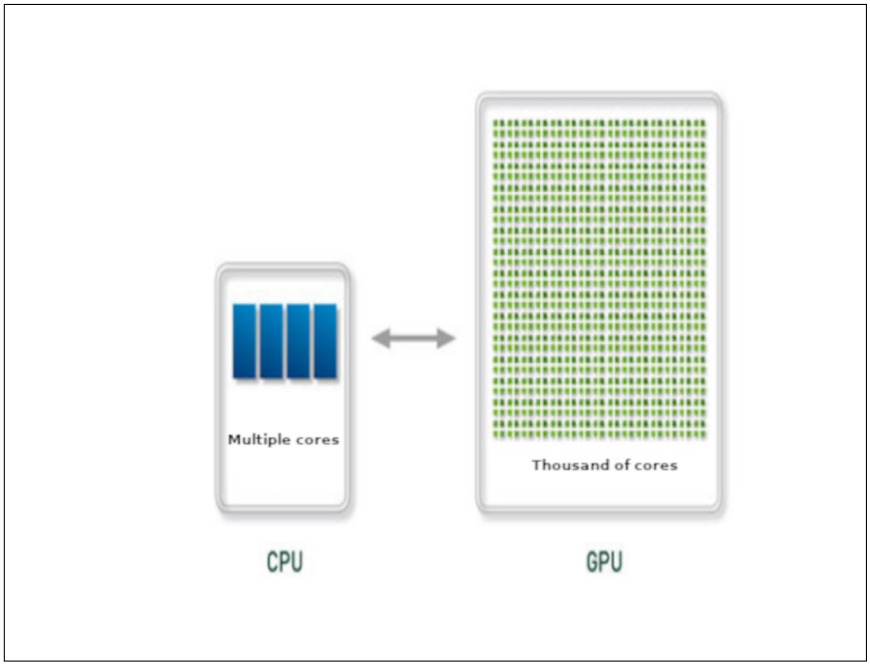

Figure 2.5 – *CPU vs GPU cores comparison*

2.7 GPU ARCHITECTURES

General Purpose GPU (GPGPU)

A Graphics Processing Unit (GPU) is massively multi-threaded architecture, initially designed for display process, visual computing, and computer games, but have recently been successfully used to accelerate many scientific applications, ranging from artificial intelligence, and smart cars to medical imaging, just to name a few. The GeForce 256 card was the world's first GPU card introduced by Nvidia in 1999, then the rival vendor "ATI" gave its first graphic card "Radeon 9700" in 2002, and used the term of Visual Processing Unit or VPU instead of GPU (Saadi *et al.* 2017).

GPGPU (General Purpose GPU) is the use of a GPU for processing non graphical entities, and for general purpose scientific and engineering computing. The best model for GPGPU is to use a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU (Saadi *et al.* 2019b). In the last years CPU is capable of carry on more numbers of cores than before. But, processing huge data in parallel needs a huge number of cores, this can be done powerfully by the GPU architecture with thousand of cores. Figure 2.5 illustrates the difference in number of cores between the two architectures. The CPU cores can be used to handle various tasks in parallel because it has large memory cache and important control units, but it has few arithmetic units (ALU). Contrarily, GPU can massively calculate in parallel a small and independent data because GPU is equipped with a huge number of arithmetic units (ALU), but it has a small memory cache and small number of control units. Thus, GPU is designed for huge data parallelization and CPU for task parallelization.

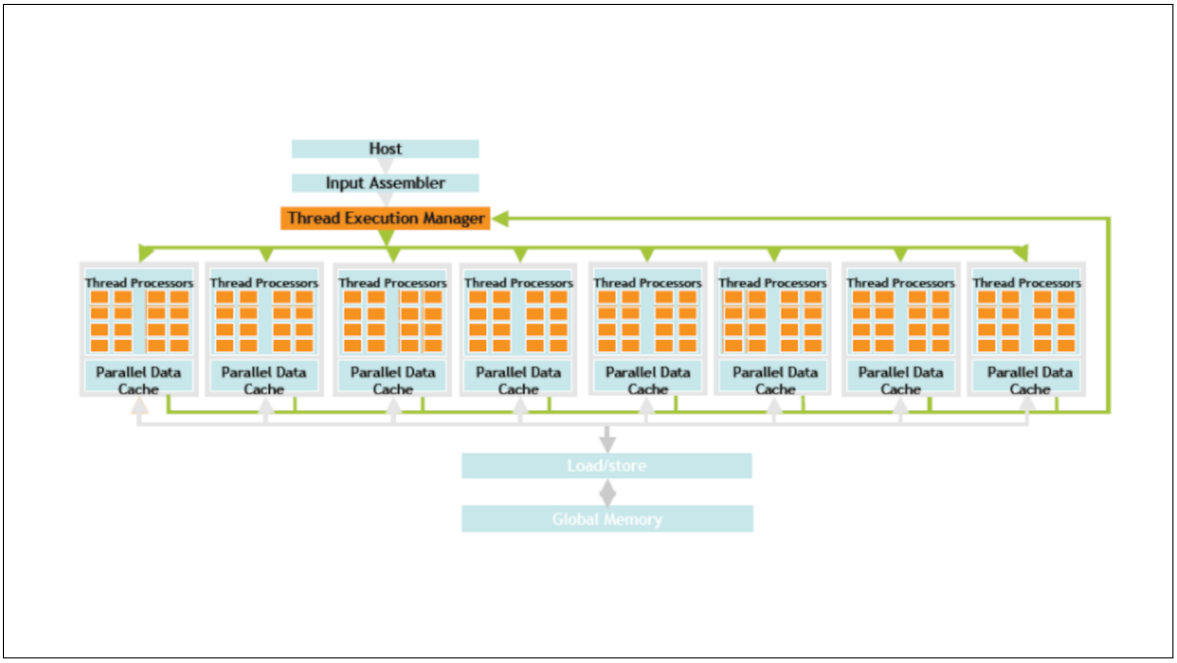

Figure 2.6 – *GPU architecture*

An effective GPGPU programming needs a good understanding of the GPUs architecture and the hardware mechanisms.

2.7.1 Modern GPU Computing Architecture

Modern GPUs are composed principally of streaming multiprocessors (SMs) (Figure 2.6 and figure 2.7), each SM consists of a number cores called streaming processors (SP), thread scheduler, registers, double-precision unit(s), and special functional units. The GPU core has the floating point and the integer arithmetic units (AUs), where instructions of the parallel program are executed. These GPU cores support multithreading, and commonly tolerating 32 to 96 threads in the current GPU architectures (Shi *et al.* 2018).

Blocks are used to organize SMs. In a typical GPU architecture a building block is composed of to two SMs; nonetheless, this number of SMs may vary from one generation to another generation of GPU card (Ghorpade *et al.* 2012) .

The GPU memory architecture can be grouped into two categories : on-device memory and on-chip memory (see Figure 2.8). Global memory, constant memory and texture memory are DRAM memory, and they belong to the on-device memory. All threads have access to global memory, while constant and texture memory are principally created for graphical computations. The on-chip memory is composed of many physical components, including register, shared memory, L1/L2 cache (Shi *et al.* 2018). The GPU global memory has a double data rate (GDDR) memory, this memory is unlike the CPU DRAM. For graphic uses, it is used as a frame buffer to hold video and images, and texture information for 3D rendering. However, for computing and for massively parallel applications, it works as very-high-bandwidth memory, nerveless with long latency than typical CPU memory. In this case the higher bandwidth compensates the

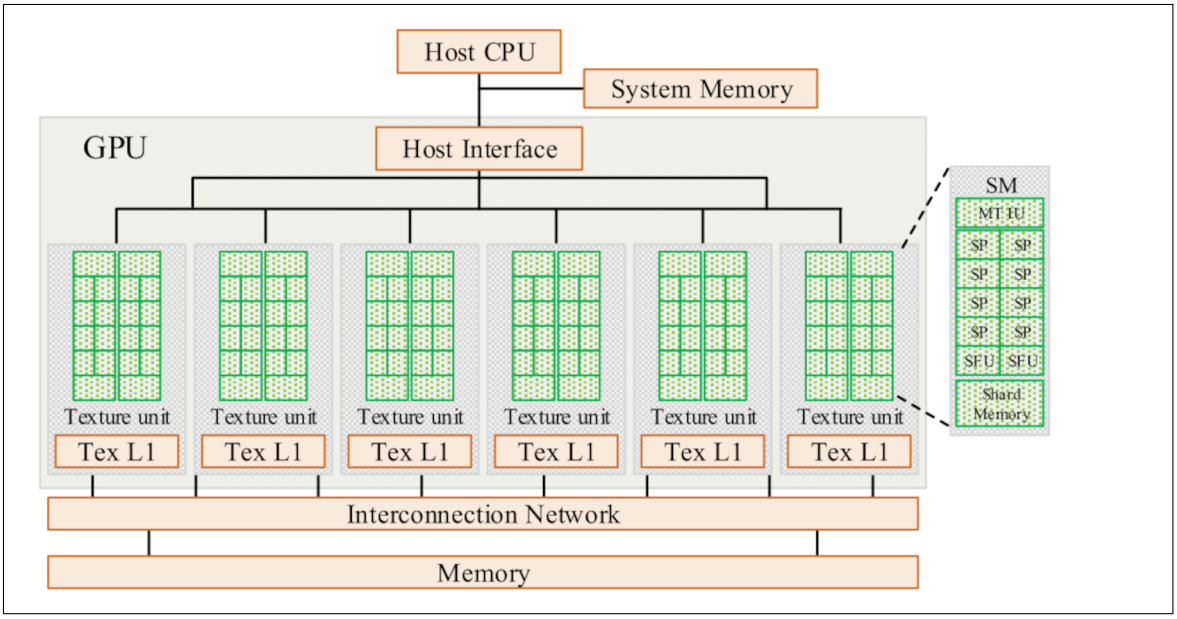

Figure 2.7 – *Modern GPU architecture (NVIDIA 2017)*

latency. Cache memories are used on a SM to reduce latency on the GPU architectures, like the constant and texture caches, and L1/L2 cache. The shared memory on each SM is small (48KB per SM in NVIDIA K40 GPU) but it is of high-speed, and is only accessible to the threads generated on that SM.

A GPU is connected to the host by the PCI-Express bus, and by the VLINK bus in the latest GPU architectures. The data is transferred between memory in GPU and the main memory in CPU usually by using the programmed DMA, which operates concurrently with both the host CPU and the GPU computing units.

More detailed information about the latest NVIDIA GPU architectures is given in NVIDIA white-paper site (Whitepaper).

2.7.2 GPU programming

There are multiple SDKs and APIs available for the programming of GPUs for general purpose computation that is basically other than graphical purpose for example NVIDIA CUDA, ATI Stream SDK, OpenCL, Rapidmind, HMPP, and PGI Accelerator. Selection of the right approach for accelerating a program depends on a number of factors, including which language is currently being used, portability, supported software functionality, and other considerations depending on the project (Ghorpade *et al.* 2012).

An overview of CUDA Programming Model

CUDA is an acronym for "Compute Unified Device Architecture". It is a popular heterogeneous programming model invented by NVIDIA company in late 2006 and designed for general purpose programming on GPUs (GPGPU) (GPGPU 2017). It allows developers and scientists to parallelize and speedup their applications. The architecture of this parallel computing model is based on a pseudo assembly language called PTX

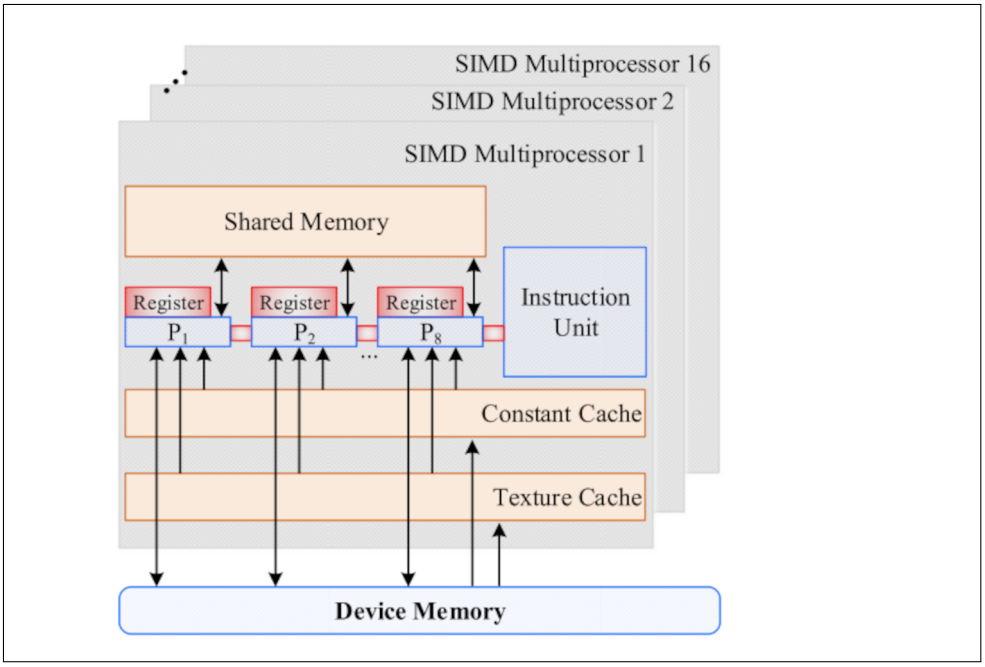

Figure 2.8 – *GPU memory architecture (NVIDIA 2017)*

(Parallel Thread Execution) and the NVCC compiler. CUDA allows the exploitation of both the CPU and the GPU in the same code. In this model, the CPU is called ***host***, and the GPU ***device***. The code is partitioned into two sections: the first section is sequential and keeps running on the CPU while the second named ***kernel*** is executed in parallel on the device by many GPU ***threads*** (Sanders & Kandrot 2010).

A thread is the basic execution unit on the GPU card (Fang *et al.* 2014). Each thread has its unique and own identifier (id). Threads are grouped into blocks which in their turn are assembled in a grid. The blocks of a grid are managed by the hardware and can be executed in any order. The number of threads per block is defined by the programmer. The threads are generated by the hardware while launching the kernel. A ***warp*** is a scheduled unit. It is a batch of 32 threads executed together by the device. Threads of the same block are scheduled warp by warp and can cooperate together using specific functions and memories. NVIDIA GPUs architecture is based on two main type of components: Streaming Multiprocessors (SMs) and memories (Nickolls & Dally 2010). SMs has numerous Stream Processors (SP). Each of them performs computations with SIMD (Single Instruction Multiple Data) parallel model. Any SP has its own registers, control units, caches, and execution pipelines. GPUs memories are mainly grouped in three categories:

- Global memory: this memory is the interface between host and device; it is private to each GPU, nonetheless shared for all SMs within the same GPU.
- Shared memory: is faster but smaller than global memory. It is a private memory space for each block and can be used only by threads within the same block. Frequently used data can be stored in this memory to accelerate computations.

- Registers: are a very fast type of memory that are private to threads.

To write a good parallel code to leverage the latest NVIDIA architectures, developers should follow the best practice guidelines and recommendations described in details in *CUDA C Programming Guide* (Corporation 2018) and in *CUDA C Best Practices Guide* (NVIDIA CUDA 2017), those recommendations are applicable to all NVIDIA GPU architectures. Further, there are guides to tuning the parallel code for each specific new architecture created by NVIDIA such as Kepler, Maxwell and Pascal. The best recommendations are summarized along these lines:

- Find good approaches to parallelize sequential code.
- Maximize GPU utilization by fin-tuning Kernel launch configuration.
- Decrease data move from the host to the device and vice versa.
- Enhance data accesses to global memory by saving data in coalesced way.
- Reduce redundant accesses to global memory by using the other kind of memories available on the device.
- Avoid threads divergence of the same wrap.

OpenCL

OpenCL (Open Computing Language) is an open, free and standard programming language (Trobec *et al.* 2018). It can be used to write programs for heterogeneous platforms consisting of CPUs, GPUs, or other types of processors. OpenCL was released in 2009 by the Khronos Group. The code written with this language is compiled at the run-time and execute in FIFO queue, thus it can work on many different platforms, and support multiple devices simultaneously.

OpenCL has one significant drawback, it is difficult to learn. Programmers need to clearly understand three basic concepts: device memory models, heterogeneous systems, and execution model.

- Memory model: the data can be placed in one of four memory types: global memory, local memory, private memory, or constant memory. The location of the data in one of these types determines how the data is shared within a work-group, and the speed to access to these data.
- Execution model: Kernels are executed by one or many work-items(threads), each work-item executes the same kernel function. Workitems are grouped into work-groups (blocks) and each work-group executes on a compute unit. Kernels are invoked over an index space called NDRange which defines the total number of work items that can be executed in parallel.

Table 2.1 shows some of the comparable CUDA and OpenCL terms.

Table 2.1 – *Comparison between CUDA and OpenCL terms.*

CUDA	OpenCL
Thread	Work-Item
Thread Block	Work-Group
global function	kernel function
Shared Memory	Local Memory
Local Memory	Private Memory
device function	Implicit
shared variable	local variable
threadIdx	get local id()
gridDim	get num groups()
blockDim	get local size()
blockIdx	get group id()
syncthreads()	barrier()

OpenACC

OpenACC is a high-level programming model based on directive (pragmas) (Chandrasekaran & Juckeland 2017). Programming with this model requires significantly less effort than using a low-level model such as CUDA or OpenCL. The compiler runs the code on the specified hardware platform at the time of compilation and handles most of the complex details of the code translations. This allows scientists to focus on their science rather than spending time to understand the target architecture details and leaning low-level languages. OpenACC offers portable code for more than one platform such as X86, multicore processors, GPUs , field-programmable gate arrays (FPGAs)...etc. The code can be compiled simply in a serial manner on CPU, ignoring the directives and producing correct results, or runs in parallel. The OpenACC compiler interprets directives and generates corresponding parallel code. If the OpenACC compiler is disabled or does not exit, the compiler ignores the directives and generates a serial program without any parallelism. The following code shows the how to write an OpenACC directives :

#pragma acc <directive> [clause, clause, . . .] new-line

OpenACC has three advantages. First, using a working program with just adding directives to it is more easy than rewriting in a new language. Second, it is designed for writing more scalable programs by allowing fewer way to insert synchronization into the code. Third, OpenACC ensures portability across different target HPC systems, programmer does not need to re-tune his code every time a vendor comes out with a new model (Farber 2016).

2.8 Parallel metaheuristics

2.8.1 Introduction to metaheuristics

Metaheuristics are approximate methods to solve complex and NP-hard problems. Indeed, these methods help to find good quality solutions in reasonable computation time compared to exact methods, but with no guarantee to reach the optimal solution (Mehdi *et al.* 2013). Metaheuristics can be devised into two categories according to the number of solutions considered at each iteration; Population-based metaheuristics (P-metaheuristics) (Mehdi *et al.* 2013), and solution-based metaheuristic (S-metaheuristics). P-metaheuristics begin with a set of solution (population) and release an iterative operation to enhance the quality of the current population, this operation is repeated until meting a stopping criteria. The most known meta-heuristic of this class are Evolutionary Algorithms (EA) like Genetic Algorithm, swarm optimization like Particle Swarm Optimization (PSO) and Ant Colonies (AO) . A survey of these methods is available in (Bozorg-Haddad *et al.* 2017).

Otherwise, in S-metaheuristics, the search method begins with a unique solution (defined randomly), this first solution is iteratively improved by exploring its neighborhood. Local search methods fall in this class. Examples of local search are, simulated annealing, tabu search, iterated local search, variable neighborhood search. A survey and a state-of-the-art of these algorithms can be found in (Trott & Olson 2010) and (Talbi 2009). P-metaheuristics are good for exploration of the global search space with a better diversification of solutions. S-metaheuristics are good for intensifying the search for solution in local region, it is better for exploitation.

2.8.2 Parallel Models of Metaheuristics

Metaheuristics are known to achieve satisfying results in an acceptable time when dealing with optimization problems. However, they suffer from the lack of scalability. it becomes limited with complex and high dimensional optimization problems. To reduce this limitation, parallel computing appears as a strong alternative. Thanks to this technology, parallel metaheuristics can obtain better results in terms of computation, and sometimes in term of the quality (Essaid *et al.* 2019), in the next section we detail the taxonomy of parallel metaheuristic.

Classification of Talbi et al.

According to Talbi et all, three parallel models for metaheuristics can be classified as illustrated in Figure 2.9; algorithmic-level , iteration-level and solution-level (Talbi 2009).

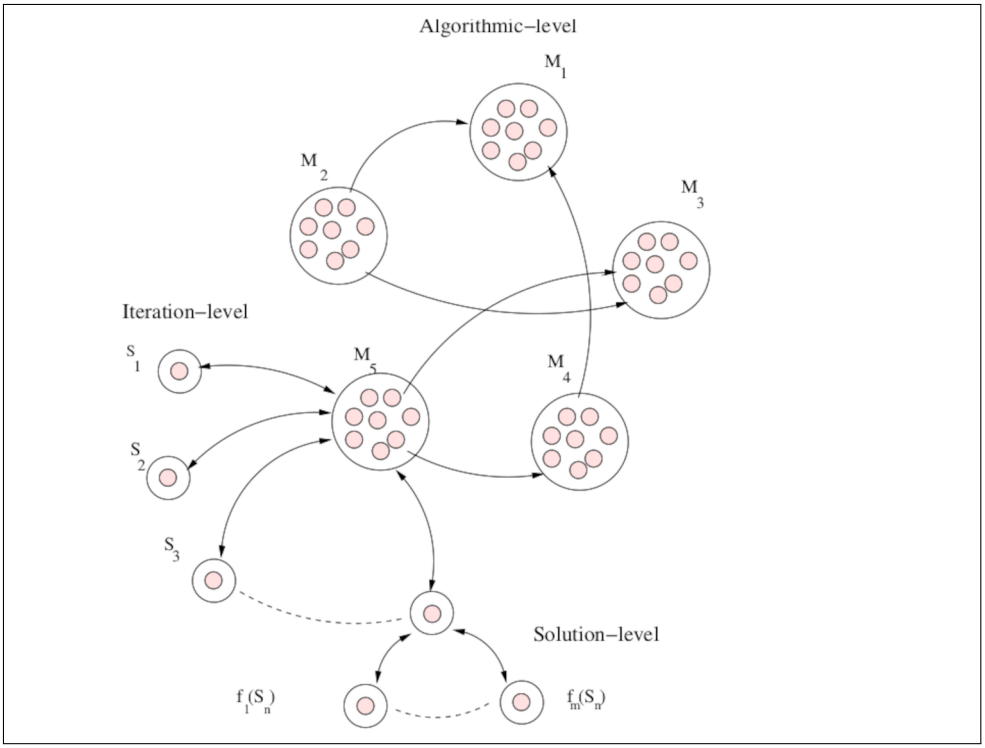

Figure 2.9 – *Parallel models for metaheuristics*

1. **Algorithmic-level:** Many metaheuristics run in parallel to enhance performance. They can be homogeneous or heterogeneous, they may start with identical or different solutions, with the same or different parameter settings for its configuration. The parallel metaheuristic launched can progress independently or in cooperative manner. For the first case, the execution time is reduced, but the quality is the same. However, in the second case (cooperative parallel models), the metaheuristics cooperate with each other to find better solutions and then improve the quality.

2. **Iteration-level:** this class is low level master-worker model, this level focuses in the parallelization of each iteration. In s-metaheuristics, the neighborhood is decomposed into smaller partitions, then each partition is processed and evaluated in a parallel. The same method is applied for p-metaheuristics, but population is decomposed into smaller population instead of neighborhood in s-metheuristic, each s-population is evaluated in parallel. In this parallelization model the behavior of the metaheuristic is not changed, the objective is to enhance performance.

3. **Solution-level :** this level focuses in the parallelization of each solution, This model is used to parallelize the objective function or constraints which are the most time consuming part. The objective is to enhance performance, indeed the quality is not altered. This parallelization level is specifically interesting when the scoring function can be itself parallelized. And this function can be divided in a number of partial functions.

2.8.3 State-of-the-art of parallel metaheuristics on GPUs

This section summarizes the sate of art of the GPU-based metaheuristics for solving some known discrete and continuous problems. This state-of-the-art is based on the recent works published by Essaid et *al* (Essaid *et al.* 2019).

GPU-based metaheuristics for discrete problems

Table 2.2 summarizes the results and characteristics of the implementations of single solution based metaheuristics (s-metaheuristics) for discrete problems such as Qquadratic assignment problem (QAP), Multi-criteria vehicle routing problem (MVRP), Resource constrained project scheduling problem (RCPSP), Multidimensional knapsack problem (MKP), the algorithms used are tabu search (TS), greedy randomised adaptive search procedure (GRASP), and simulating annealing (SA)

Table 2.2 – *GPU based S-Metaheuristic for discrete problems.*

Algorithm	Problem	Parallel level	Acceleration	Benchmark	CPU/GPU
TS	QAP	Algorithm + Iteration level	420x vs CPU and 70x vs 6 core CPU	QAPLIB	CPU: Intel 6 cores I7 / GPU: Nvidia GTX 480 with 480 Cores
TS	QAP	Algorithm + Solution +Iteration level	GPU saves up 3h compared to CPU	QAPLIB	CPU: Intel I5 / GPU: Nvidia GTX -680 with 1536 Cores
TS	MVRP	Algorithm level	14x vs CPU sequential implementation	Instance of size from 10 to 500	CPU:Tesla S2050 , GPU: Nvidia GTX 480 with 480 Cores
TS	RCPSP	Algorithm + Iteration level	5.4x vs parallel CPU and 22 x vs sequential CPU	J30, J60, J90, J120	CPU: AMD X4 945 server, GPU: Nvidia GTX -650 with 768 Cores
GRASP	MKP	Algorithm + Iteration level	No acceleration but quality improved	ORLIB	Not motioned
SA	TSP	Algorithm + Iteration level	14.84 vs CPU implementation	9 instance from TSPLLIB	CPU: AMD A8 3870k, GPU: Nvidia GTX -680 with 1536 Cores

Table 2.3 outlines some works of parallel p-metaheurstic on GPU such as Ant colony optimization (ACO), Particle Swarm Optimization (PSO), and evolutionary algorithms for solving discrete problems on GPU, the problems tackled are Unmanned Aerial Vehicles (UAV), Vertex Coloring Problem VCP, edge detection in images, simulate the pedestrian movement, the Satisfiability Problem (SAT), max constraint satisfaction problems (Max-CSPs), Golomb ruler problem, massively multi-modal deceptive problem (MMDP), subset sum problem (SSP), and Maximum CUT problem (MAXCUT), task scheduling problem...etc

Table 2.3 – *GPU based P-Metaheuristic for discrete problems.*

Algorithm	Problem	Parallel level	Acceleration	Benchmark	CPU/GPU
EA	Golomb ruler	Solution + Iteration level	10x	8 instances of different length of ruler	CPU: Intel 64 (2.4GHz),GPU Nvidia Tesla M2090 512 cores
SS	Golomb ruler	Solution + Iteration level	1.22x	8 instances of different length of ruler	CPU: Xeon E5645(2.4 GHz),GPU Nvidia M2090 512 cores
ACO	TSP	Solution + Iteration level	546,66x	TSPLIB	CPU:Intel i5(3 GHz), GPU Nvidia GTX 680 with 1536 Cuda cores
ACO	The pedestrian movement	Solution + Iteration level	18x	2D grid and agents manually generated	CPU: Intel i7(2,8 GHz),GPU Nvidia GTX 560ti with 448 Cuda cores
ACO	Vertex coloring	Solution + Iteration level	19.68 to 36.81x	6 instance from [22]	CPU:Intel i7 4790(3.66 GHz),GPU Nvidia GTX 1080 with 2560 Cuda cores
ACO	MKP	Solution + Iteration level	575x	ORLIB	Not montioned
ACO	Edge detection	Iteration level	150x	The standard test images [22]	CPU:Intel i7 950 (3,06 GHz), GPU Nvidia GTX 580 with 580 Cuda cores
ACO	SAT	Solution + Iteration level	21x	Instance from the SAT VNFs	CPU: Intel i7 3770K (4.5 GHz),GPU Nvidia GTX 680 with 1536 Cuda cores
SNS+uGA	MMDP and MAXCUT	Iteration level	216x	Large instance K=20,30,40	CPU: Intel i7 920 (2.66 GHz), GPU Nvidia GTX 650 with 384 Cuda cores
Graphic cell	Task Scheduling TS	Solution + Iteration level	Not mentioned	Instance as in [22]	CPU: Intel Xeon E5440,GPU Nvidia Tesla C2050 GTX 580 with 448 Cuda cores
Memetic Algorithm	TS with precedence relations	Algorithm + Iteration level	696x	Benchmark as in [22]	CPU: Intel Xeon (2.8 GHz),GPU Nvidia GTX 480 with 480 Cuda cores
RRLS	Probabilistic TSP with deadlines	Solution + Iteration level	10x	Instance chosen as in [22]	CPU: quad core Intel i7 950 (2 GHz), GPU Nvidia GTX 580 with 512 Cuda cores
GA	QAP	Iteration level	30x	10 instance from QAPLIB	Not montioned
PSO	MKP	Solution + Iteration level	3.5 to 9.6x	Taken From [22]	CPU: Quad core Intel i7 920 (2,66 GHz), GPU Nvidia GTX 580 with 512 Cuda cores
2 variants of PSO	Max-CSPs	Solution + Algorithm Iteration level	1,3x vs GPU-PSO	Randomly generated in stances [22]	CPU: Intel i7 2630QM (2.0 GHz), GPU Nvidia GTX 525M with 96 Cuda cores
ABC	connected cover sensors	Iteration level	5x	100*100 square area	CPU:i3 (3.4GHz), GPU Nvidia GTX 760 with 1152 Cuda cores

GPU-based metaheuristics for continuous problems

This subsection outlines different implementations of GPU-based metaheuristics addressed to solve continuous problems (Essaid *et al.* 2019). Table 2.4 summarize the characteristics and the results of the implementations of the important works to accelerate different metaheuritics such as PSO, Differential evolution (DV), memetic algorithm, Genetic Algorithm (GA), CU bee algorithm (CUBA), multi-objective tabu search (MOTS2)...etc.

Table 2.4 – *Characteristic of GPU based Metaheuristic on continuous problems.*

Algorithm	Problem	Parallel level	Acceleration	Benchmark	CPU/GPU
DE	Iteration level	4.475x	Function from CEC 2008	Improved	CPU: intel core quad GPU: Nvidia GeForce GTX285 with 240 cores
DE-BSA-SA	Solution + Iteration level	40x	Function from [22]	No improvement	CPU: intel I5 GPU: Nvidia GeForce GTX680 with 1536 cores
DE+PSO	Solution + Iteration level	Not mentioned	4 test from CVSSP+ 15 images of hippocampi	PSO performs better in human body pose but DE is better in hippocampus localization	CPU: intel I7 GPU:Nvidia GeForce GTS450 with 198 cores
MA-SW chains	Solution + Iteration level	82.17x	CEC 2010	No improvement	CPU: intel I7 GPU: Nvidia Titan with 2688 cores
PSO	Solution + Iteration level	46x	Sphere, Rosenbrock, rastrigin, Griewank, ackley, de jong, Easom	Quality improved in sphere and Griewank	CPU: intel I7 GPU: Geforce GTX 980 with 2048 cores
GA	Iteration level	1.18 to 4.15	Function taken from [22]	No improvement	CPU: intel I5 4200 /GPU: Geforce GTX 740 with 384 cores
CUBA	Algorithmic + Iteration level	18x	Function taken from [22]	Quality improved	CPU: AMD Athlon II/ GPU: Geforce GTX 460 wth 336 cores
MOTS2	Solution + Iteration level	23.7x	ZDT and ZDT2 function	No improvement	CPU: not mentioned/ GPU: quadro 1000m with 96 cores

Discussion

Few works use s-metaheuristics to solve continuous problems. Iteration level model is used to implement and adapt s-metaheuristics on GPU, this parallel model is generally used to accelerate the generation and the evaluation of neighbors without affecting the behavior of the algorithm (Essaid *et al.* 2019). Algorithmic level is used to enhance the exploration

of the search space by launching multiple instances. In this model, communication strategies like diversification strategy can be adopted between instances to enhance quality by better exploring the area to find promising search regions. Nevertheless, beside the improved quality of this model, lot of works show that a performance is not achieved if we consider the huge computing power of GPU. So it is a compromise between performance and quality (Essaid *et al.* 2019).

For p-metaheuristics with discrete problems, and according to (Essaid *et al.* 2019), iteration level is the most adopted parallel model because of the large number of individuals to be evaluated and processed. However, algorithmic level is used only in few papers, for example the swarm is partitioned into sub swarms, then each sub swarm runs a metaheuristic separately. To further speed up the algorithm, the solution level used to generate and evaluate solutions, for example in ACO metaheuristic, tour construction phase can be performed in parallel, where an ant generates a set of possible moves in parallel. Then, it selects the best move. Always according to (Essaid *et al.* 2019), solution quality has been treated in some works with a small speed-up factor of at best 1.19x. While in other works the quality is not improved but a significant acceleration factor of at best 696x is achieved. Further, few works made the exception, they improve the quality along with keeping a high speed-up.

P-metaheuristics have been widely implemented to tackle continuous problems. Besides iteration level, in most of works, solution level model is used to enhance the speed up of the metaheuristic. The quality is maintained, and improved in some works. According to (Essaid *et al.* 2019) algorithmic level is less implemented in this case. This level is implemented by running sub-populations with defined algorithm and an appropriate communication strategy.

Finally, the common point to design an efficient parallel metaheuristic on GPU is to maximize data parallelism to enhance the occupancy of the GPU, and then to benefit from the computation power of the GPU. Moreover, GPU impact becomes clear when dealing with large instances.

2.9 Challenges and guidelines for porting metaheuristics to GPU

The parallel models of metaheuristics need adaptation and modification with huge effort at the design and implementation level to leverage the new requirements of the GPU architectures. Many challenges and issues have to be taken into account. These issues are mainly related to the latency of the GPU memories and their hierarchical memory management, divergence of GPU threads and their synchronization, optimization of data transferring between the CPU and GPU, the efficient distribution of date processing between the host and the device (Mehdi *et al.* 2013).

2.9.1 Memory management and data placement

To design an efficient parallel metaheuristic on GPU, a deep study has to be performed on both metaheuristic (p-metaheuristics or s-metaheuristics)

and their data structures, and the GPU memories. P-metaheuristics need the exploration of a large number of individuals to diversify the search, s-metaheuristics requires exploring large neighborhoods, so the data size and the access frequency are different. On the other hand, these data have to be mapped to deffirent kind of GPU memories with different sizes and access latency. For example, registers and shared memories are very fast but very small, global memory is larger in size and relatively slower. Thus, during the execution of metaheuristics on GPU, the different threads may access multiple data structures from multiple memory spaces. Programmers have to take into account this point to efficiently place the different data structures of the metaheuristic on the different memories to benefit from both the faster memories and the larger ones (which data will be placed on which memory). Generally, the most accessed ones should be put on faster memories (registers, shared memory) and larger ones on the lager memories (global memory). An other key is to map efficiently threads with the corresponding metaheuristic individual, for example one neighbor per thread for s-metaheuristic, or single population per threads block for p-metaheuristic, etc.), this parameter must be well defined to maximize the occupancy of the GPU and to optimize memory access times and CPU/GPU communication. The high occupancy covers the high latency of the GPU memory, it can be ensured by Increasing the size of neighborhood or population for s-metaheuristics and p-metaheuristics respectively. However, the bad use of the high occupancy can deteriorate the performance. For example, when increasing the number of thread until it becomes bigger than registers capacity, then, this would require using global memory, which is a very slow memory.

The coalesced access to the memory is another important issue to deal with to optimize the performance of GPU-based metaheuristic; the global memory is optimized for coalesced accesses. However, the constant and the texture memory are uncoalesced and read-only memories; they are optimized for simultaneously accesses of concurrent threads to the same location. Therefore, coalesced data have to be placed on global memory, concurrent data on constant memory (e.g. constant for fitness evaluation), and uncoalesced read-only data on texture memory. The shared memory should be used for the most accessed data structures (e.g. population of individuals).

The data transferring between GPU and CPU, is another important factor, good strategies should be adopted, for example using the so called Page Locked Memory. It will disable the memory paging, and use Write Combining Allocation. It disables CPU caching of a memory that the CPU will only write to, and improves transfer performance by 40 %.

In addition to the memory access and latency issues, the encoding step is crucial when designing a metaheuristic for a specific problem. Indeed, the data types should be chosen carefully, the ranges of this data values should also be analyzed to minimize the memory space to be occupied by this data.

2.9.2 Threads divergence and synchronization

GPU architecture is based on massively parallel multi-threading. Thread synchronization issue is due to this architecture and the synchronization requirements of the implemented metaheuristic. Indeed, GPUs are based on Streaming Multiprocessor (SM), each SM contains huge number of cores executing the same instruction of different threads following the SIMD model. These threads must be executed in a warp with different data elements. An efficient parallelization on GPU is achieved when running a large number of threads (thousands of threads). Nevertheless, the order in which these threads are executed is not known. To face this challenging issue the programmer has to manage explicitly the threads by the insertion of barrier synchronizations in the codes or by using other methods to avoid concurrent accesses to data structures.

In Thread divergence issue, the challenge is to redesign the traditional irregular metaheuristic codes to eliminate these divergences (Cecilia *et al.* 2013) (Delévacq *et al.* 2013). Indeed, metaheuristics contain irregular loops and conditional instructions when generating and evaluating solutions in the same block. In addition, threads of the same warp have to execute simultaneously instructions directing to different branches, considering that in a SIMD model threads of a same warp execute the same instruction at a time. Therefore, the different branches of a conditional instruction which is data-dependent force to a serial execution of the different parallel threads of the same wrap, which deteriorates considerably the performance of the parallel metaheuristic.

2.9.3 Code mapping and CPU/GPU communication

To achieve the best performance of parallel metaheuristics on GPU in terms of execution time, an efficient decomposition of the code and the efficient re-partition of taskes between the CPU and the GPU should be adopted. The objective is to leverage the GPU computing power without losing performance in CPU/GPU communication and memory accesses. Thus, the code should be carefully analyzed in order to decide which part of the code will be stay and executed on CPU, and then which compute-intensive part will be send to the GPU for parallel execution.

CPU/GPU communication is another important factor to take in consideration when designing parallel metaheuristic, this communication is done through the global memory which is relatively a slow memory making the memory transfer between the CPU and GPU time-consuming and can significantly degrade the performance of the application. Therefore, programmer should optimize the code to minimize the amount of transferred data between the host and the device, and the accesses way to these data, these technique participate in the reduction of the whole execution time of the metaheuristic.

Several approaches can be used to analyse GPU code (kernel) bottlenecks, CUDA profiler (Matloff 2006) is a good tool which allow to identify if a kernel is limited by bandwidth or by the arithmetic operations, and locate the part of the code that may lead to better performance.

2.10 Related Works solving the MD with parallel meta-heuristic on GPU

In this section, we focus on the existing works on solving the molecular docking problem with parallel meta-heuristic on GPUs. Parallel optimization of molecular docking has turned to take advantage of the GPU. For example, to accelerate the calculation of the energy of interaction (scoring function) on GPUs. The latter consumes up to 80% of the total execution time, thus, it is an important bottleneck (Dong *et al.* 2018).

Sukhwani et all reached a speedup of at least 17.7x with respect to one CPU core and 6.1x with respect to four CPU cores (Sukhwani & Herbordt 2009). The ligand and receptor data are transferred to the device memory, the receptor data are stored in the global memory. The large ligands are also stored in global. However, the smaller ligands are stored in the shared memory to benefit from the rapid access time from this kind of memory, and consequentially enhance the performance. To perform the scoring function, the constant memory is used to store the scoring coefficients. The first scheme propose to divide and distribute the work to K different blokes to compute its respective best scores, the top score of each bloks is stored in the GPU global memory. The results was inspected and gives a significant slowdown compared to the CPU code. In the second scheme, aothors assigned different tasks to different thread with, each block containing M threads (M = 500). The best score found by each thread is stored in a shared array in the GPU shared memory. The master thread (thread 0) processes the shared array to find the best scores and stores it in the global memory.

Gine's D. Guerrero et al. developed the algorithm based on CUDA programming model for calculating the energy of interaction (Guerrero *et al.* 2011).Each atom from the receptor molecule is represented by a single thread. Then, every CUDA thread goes through all the atoms of the ligand molecule. Each thread simply charges with the electrostatic interaction calculations with its corresponding atom of the receptor molecule and all the ligand molecule atoms. In order to avoid communication overhead, each thread block should contain all information related to the ligand and the protein. They tested the kernel on GPUs and got a speedup of 260 times.

FlexScreen is parallel docking software, in which the authors designed a hybrid parallel architecture of MPI-OpenMP for CPUs, and CUDA for GPU. MPI is used to send data to nodes, then energy calculations are carried out on each node with OpenMP. Experiments showed the CUDA program on GPU outperforms the MPI-OpenMP hybrid parallel program in terms of speed (Guerrero *et al.* 2012).

MolDock (Simonsen *et al.* 2011) is a flexible molecular docking software with uses modified differential evolution search algorithm, this software utilizes CUDA for GPU and multithreading for CPUs. This parallel strategy can be applied in other flexible molecular docking algorithms.

GeauxDock uses Monte Carlo metaheuristic with a new hybrid fitness function combining physics-based and knowledge-based energy calculation. The parallel optimized version of GeauxDock (Fang *et al.* 2016) can runs in the multi-core CPU, as well as in GPU, or Intel Xeon Phi. All three

platforms share a common code for the user, whereas back-end codes have one version for GPU, and an other code for CPU and Xeon Phi. Geaux-Dock was designed with two level of parallelism. At coarse grain level, each protein-ligand complex conformations is assigned to different CUDA thread blocks on GPU . At the fine-grained level, the protein-ligand conformation data are accessed in parallel by CUDA threads of GPUs.

E. Vitali et al proposed a parallel model for MD on an heterogeneous system with one or more GPUs or multi-core CPU. The docking approach used is based just on geometric features of the ligand and of the target protein (Vitali *et al.* 2019). For parallellization strategy, they utilize OpenACC for the GPU, OpenMP for the CPU work sharing and MPI for the inter node work sharing and communication. They reached a 25% throughput improvement within the single node.

2.11 Conclusion

In this chapter, we introduced the basic concepts related to parallelism in a general way. We first presented the different classification of parallel architectures that exist in the literature, and the metrics used to evaluate parallel algorithms such as speedup and efficiency. After that, we detailed the GPU architectures and how scientists can leverage this recent architecture to speed up their algorithms efficiently. We then presented parallel metaheuristics and the challenge to face in order to implement these parallel metaheuristics on GPU architectures. In the last part of this chapter we detailed the literature reviews on solving the MD problem with parallel metaheuristics on GPU architectures.

In the next chapters, we will present our contributions to solve the MD problem with the parallel BSO metaheuristic and how to efficiently implement it on GPU to speed up the simulations of molecules interaction.

BSO Metaheuristic For MD problem

3

3.1 Introduction

In the first chapter of this thesis, we detailed the problem of molecular docking, after which we gave the different metaheuristics used in the literature to solve the problem, such as, the genetic algorithms GAs, taboo search TS, and particle swarm optimizations PSOs. This chapter presents our first contribution in this thesis. It deals with the molecular docking problem, which is a set of techniques used in the new drug discovery process. This method helps in screening for new drug candidates quickly with lower cost. It consists in calculating the optimum orientation, position, and conformation of a new molecule (Drug candidate) to an existing molecule (which is at the origin of a disease), to form a stable molecular complex with overall minimum energy. Since there are an infinite number of possible positions and orientations, this problem is NP-hard. This problem can be modeled as an optimization problem and our goal is to solve it in reasonable time through the use of metaheuristics. Motivated by the success and the power of Bees Swarm Optimization (BSO) metaheuristic (Drias *et al.* 2005), we propose a new docking search algorithm based on BSO metaheuristic, we called this solution BSO-DOCK (BSO for Molecular Docking).

This chapter is structured as follows. In Section 3.2 and Section 3.3, we describe biological and artificial bee swarms respectively, we introduce the

most know artificial bees swarm metaheuristics with focusing on BSO. In Section 3.4, we present our strategy to solve the docking problem with the BSO algorithm, we detail the component of this metaheuristic to efficiently model our problem. In Section 3.5, we first describe the data-set used and the experimental setup and then we present some results and discussion.

3.2 Natural bees

Bee swarms are considered to be among the most important swarms in nature. These swarms allocate tasks dynamically, and adapt to different changes in the environment in a collective and intelligent way. Bees have a photographic memory, antennas and navigation systems, which make it possible on the one hand, to detect new hives and, on the other hand, to perform their various biological tasks such as foraging and breeding. Several researchers have drawn inspiration from the collective behavior of natural bees to create models that solve combinatorial optimization problems.

Before presenting the different algorithms based on bee swarms, we will see the different types of bees such as: the queen, the workers and the male, as well as the behavior of bees to collectively perform biological activities such as nest selection, food, and marriage (wiki Bees 2020).

3.2.1 Types of bees

The queen

The queen is the only fertile female in the swarm (see Figure 3.1), so she is the mother of all the individuals who make up the community (the drones, the workers and the future queens). Its laying capacity is very high, its daily production often exceeds 1500 eggs. The queen does not have the appendages of the workers such as pollen baskets, wax secretion glands and the well-developed honey bag. Its almost exclusive food is a secretion, called royal jelly, produced by the glands in the workers' heads. The average life span of the queen is one to three years.

The worker

The workers are always much more numerous than the males. In the hive of a temperate region, the number of workers is between 8000 and 15000 in spring, but can exceed 8000 in early summer. The workers are unable to mate and therefore to reproduce. They secrete wax, build cells, collect nectar, pollen and water, transform nectar into honey, clean the hive and defend it against predators. The life expectancy of female workers is about six weeks. During the first three weeks of their lives, the workers build the combs, clean and polish the cells, feed the young and the queen, control the temperature, and evaporate the water from the nectar until it became a thick honey consistency. After this period, they will collect nectar from the flowers and defend the hive.

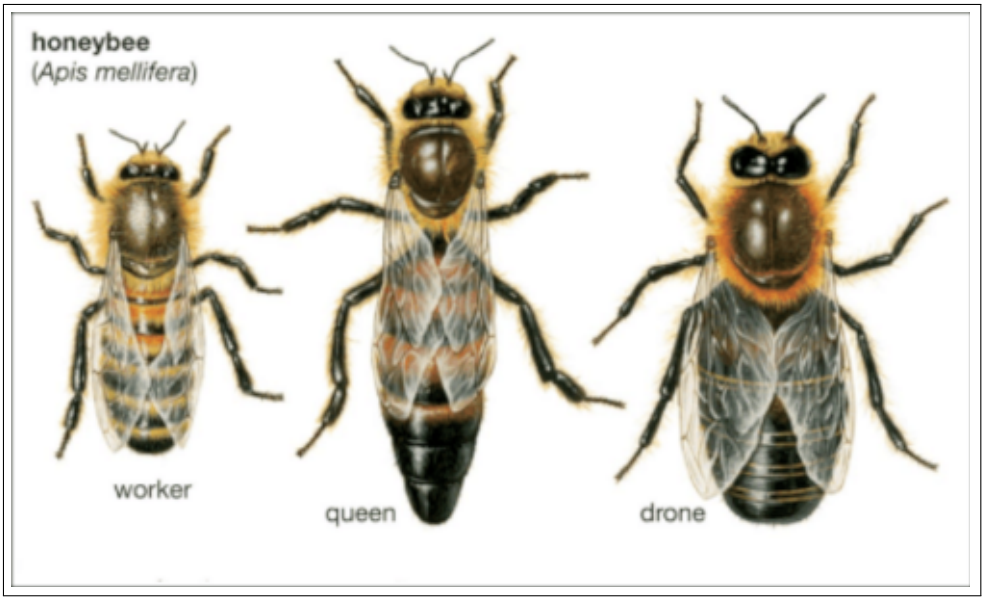

Figure 3.1 – *Types of honeybees*

The male (Drone)

It does not work; it is unable to feed himself and cannot even defend the colony, since it has no sting. The number of males is between 300 and 3000 in the colony, its only function is to mate with the queen, when the queen leaves the hive, the males are attracted by her smell. After mating, the male dies quickly; their average life span is 90 days.

3.2.2 The behavior of bees in the nature

Selection of nest location

Bee companies live in nests called hives. In the last month of spring or early summer, the queen flies away with thousands of workers, to build another colony, they are divided into two groups: the queen with half of the workers and the queen's daughters with the rest of the workers. For nest selection, some bees explore the sites; the other bees remain inactive, probably retaining swarm energy, until they make a decision and migrate to the new nest. The bees responsible for the search for food indicate the different nests by several dances called the *stirred dance*. The direction is indicated by an angle from the sun, the distance is defined by the duration of a stirred part of the dance, and the stir turns 40 to the right to indicate the food source. Through the path represented by the dance, the bee turns vertically around the center and forms an angle with its body by a vibration. This angle connects the hive with the nectar.

Food

The main role of the bee in search of food is to find a very rich source of food to obtain the maximum amount of nectar. And for that, there are different types of these bees:

Unemployed bees

There are bees with no knowledge of the food source and no view of the food source to exploit. So these bees stay in the hive and wait for the dance of the other bees to establish the food source.

Employee bees

An employee bee is associated with a particular food source. it has sufficient knowledge to know the food source. It finds and exploits the source, memorizes the location and loads part of the nectar and discharges it into the hive.

Marriage

The mating process takes place in the open air. Between April and June, the queen enters the ideal period for mating. It reduces its weight by switching from a royal jelly diet to a honey diet and prepares for nuptial flight. On a sunny day, Iit slowly slips out of the hive and flight. The meeting will take place somewhere in the sky, between the queen and males from the hive but also males from other hives. What is strange is that even during the breeding period, a meeting between the queen and the males in the hive will not produce any follow-up. It is only at the exit of the hive that the queen makes herself available for mating.

3.3 Artificial bees

There are several algorithms that simulate the behavior of natural bees. In the following we will detail the most used algorithms in the literature (Karaboga & Akay 2009) .

3.3.1 Bee Colony Optimization metaheuristic (BCO)

Initially, all bees are located in the hive. Each bee flies to perform a series of movements or displacements in order to build a step-by-step solution. Bees incrementally add components to the current solution until they obtain one or more eligible solutions (Teodorovic & Dell'Orco 2005). During the exploitation of the solution space, the bees perform two steps. In the forward step, the bees create several partial solutions based on the collective experience accomplished in previous iterations. In the backwards step all bees return to the hive and communicate with each other in order to participate in the construction of complete solutions for each iteration. Indeed, each bee shares its already created partial solution and its quality with the other bees. Complete solutions of a single iteration are designed while saving the best solution. This process must be repeated until a maximum number of iterations is reached (see Algorithm 1).

Algorithm 1 The BCO algorithm

Begin

1: Initialize the basic population
2: **while** Stopping criteria not met **do**
3: Each bee moves through the search space by building a partial solution step by step.
4: Each bee sends its partial solution and its quality to all bees.
5: Build the complete solutions for this iteration.
6: Save the best solution for this iteration
7: **end while**

End

3.3.2 Artificial Bee Colony (ABC)

Karaboga proposed this metaheuristic called Artificial Bee Colony in (Karaboga 2005), food sources represent the solution space of a given problem and the nectar from each source represents the quality of a single solution. There are three types of bees (worker bees, spectators and masters). The number of worker or spectators bees is equal to the size of the population in a single iteration.

First, the ABC algorithm randomly generates the initial population that contains **SN** solutions such that **SN** is the number of worker or spectator bees. Then, each worker bee establishes modifications on the current solution according to the quality of the solution and the local information in its memory. If the quality of the new solution is better than the previous one, then the bee will replace the previous solution with the new one. Afterwards, the worker bees share their solutions with the spectator bees. Thereafter, each spectator bee chooses the solution closest to the current solution. As in the case of worker bees, each spectator bee establishes a modification on the current solution, evaluates the found solution and keeps the best solution among the two solutions. At the end, the master bees explore other research regions, each time saving the best solutions produced by the worker and spectator bees. This process must be repeated for a maximum number of iterations or until the stopping criterion is met (see Algorithm 2).

3.3.3 Bee Algorithm (BA)

This metaheuristic has been introduced by Pham et al (Pham *et al.* 2005). It is inspired by the natural foraging behavior of honey bees. First, a population of **n** bees is generated, then divided into two groups. The first group contains **m** bees that explore **m** different regions of the search space, each bee explores its region independently and returns the best solution from its region. The best solutions from each region are saved at each iteration so that the best regions are likely to be explored in the next iterations. The remaining **n - m** bees establish a random search to reconstruct a new population for the next iteration. This process must be repeated up to a

Algorithm 2 The ABC Algorithm

Bging

1: Initialize the basic population
2: **while** the stop criterion is not reached **do**
3: Place the worker bees on their food sources.
4: Place the spectators bees on the solutions already found by worker bees.
5: Save the best solution found by the workers and spectators spectators.
6: Send the master bees to the solution area in order to find other sources of food
7: **end while**

End

maximum number of iterations. Figure 3.2 clearly explains the different steps of this algorithm.

3.3.4 BeeHive metaheuristic

In nature, there are two types of bees, bees at a short distance that search for food near the hive. and bees called *long-distance bees*, that explore other areas far from the hive. The algorithm **BeeHive** was inspired by this idea. Indeed, the set of bees is divided into two subgroups. The first sub-set applies a local search near the hive, while the second one applies a diversification operation by exploring other areas of the search spaces. At the end of each iteration, we save the best solutions found by the two types of bees (Wedde *et al.* 2004).

Algorithm 3 The BeeHive algorithm

Bging

1: Divide the initial population into two subgroups (small distance bees and long distance bees)
2: **while** the stop criterion is not reached **do**
3: Each short-distance bee applies a local search to its solution
4: Determines the best solution for the short-distance bees
5: Each long-distance bee explores its region locally
6: Determines the best solution for the long-distance bees
7: Save the best solution for these subgroups
8: **end while**

End

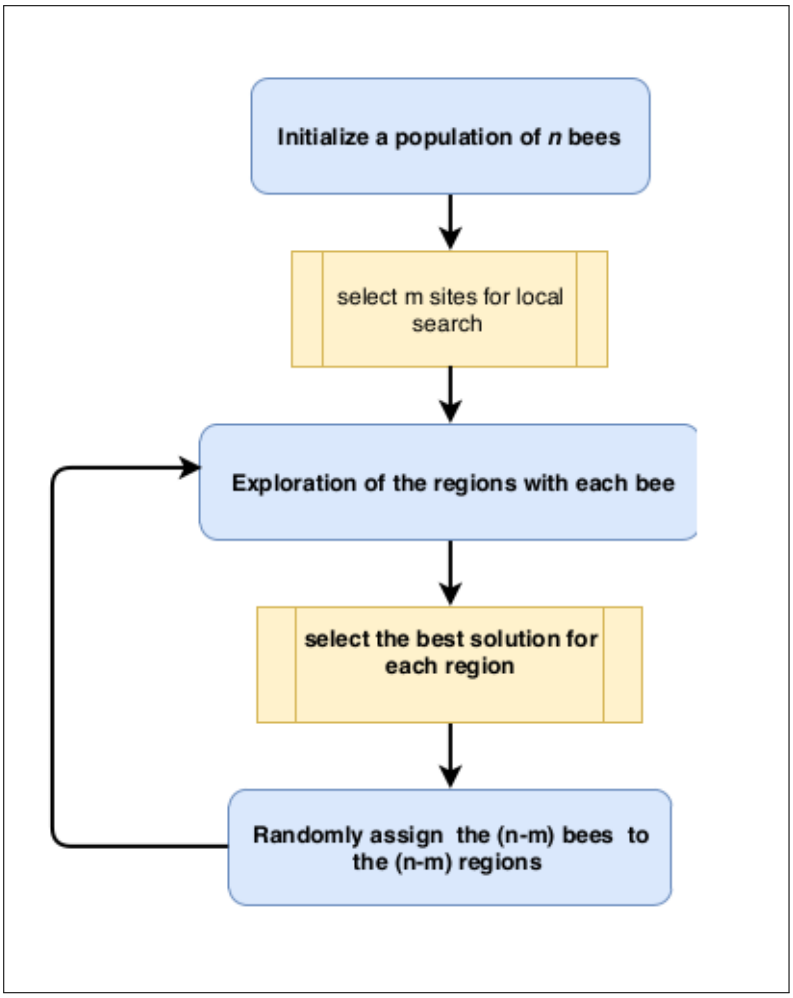

Figure 3.2 – *BA algorithm*

3.3.5 Marriage Bee Optimization (MBO)

This metaheuristic is based on the mating (marriage) behavior in honey bees (Abbass 2001). The queen is the main individual of this event, it moves with a certain energy and speed and chooses the best males. If the male is selected by the queen, production is carried out while adding his sperm to the queen's ovum. By using mutation and crossbreeding, larvae production is carried out, thereafter, each worker takes care of a single larvae. The best larvae will be the new queen of the next iteration. The following algorithm describes the structure of this algorithm.

Algorithm 4 The MBO algorithm

Begin

Random initialization of the queen
while the number of mating is not reached **do**
  Initialize the queen's energy and speed
  while The energy of the queen > 0 **do**
    The queen moves and chooses the males
    if The male is selected **then**
      Add her sperm to the queen's ovary
      Modify the queen's energy and speed
    end if
  end while
  Produce larvae with mutations and crosses
  Use workers for larval occupation by conducting a local search from each larvae
  Modify the fitness of the workers
  if The best suitable larvae are the queen **then**
    Replace the queen's chromosomes with the chromosomes of the best larva
  end if
end while

End

3.3.6 Bees Swarm Optimization BSO

The metaheuristic Bees Swarm Optimization (BSO) is among the newest swarm intelligence algorithms. It has been proposed by Drias et al (Drias *et al.* 2005). It is based on a swarm of artificial bees cooperating together to solve an NP-hard problem (see Figure 3.3). BSO simulates the collective honey bee behavior in nature when looking for nectar of the nearest and richest sources using the bee dance. In this algorithm, an artificial bee named InitBee works out to find a first solution named **Sref** with some good features. Then, it uses **Sref** as a starting point to find a group of disjoint solutions called **Search-Area** or **space of regions**, the aim is to maximally exploit the search space, by using a certain strategy in such a way that the different points are very distant from each other. Subsequently, every bee takes one solution from the **Search-Area** group and considers it as its starting point to do a local search (intensification) to

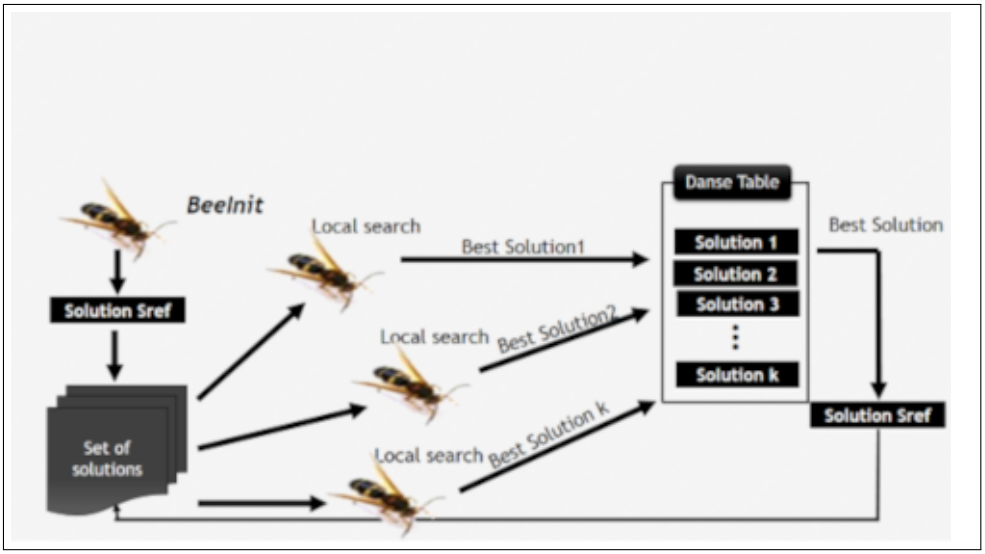

Figure 3.3 – *BSO algorithm*

look for other potential solutions. After that, every bee communicates the found solution to all its neighbors through a table named **Dance Table**. The best solution stored in this table becomes the new reference solution for the next iteration. A scoring function is used to choose the best solution.

A **taboo-list** structure is used to store the reference solution of each iteration in order to avoid cycles and prevent bees from returning to an already found solution. However, if the quality of the solution found by the bees is not improved after a given number of iterations, the bees introduce the technique of diversification, which consists in selecting from the taboo-list, the furthest solution from the current one. The algorithm stops when the optimal solution is found or the maximum number of iterations is reached.

Algorithm 5 BSO Algorithm

Begin

1: *Sref* = the solution found by InitBee
2: evaluate *Sref*
3: **while** The stopping criterion is not met **do**
4: Insert *Sref* in taboo list
5: Search-Area (*Sref*)
6: Assign **regions** to **Bees**
7: **for** each Bee **a do**
8: Local-Search(**a**)
9: Save the result in the **table dance**
10: **end for**
11: evaluate the solution (**table dance**)
12: choose the new reference solution *Sref*
13: **end while**

End

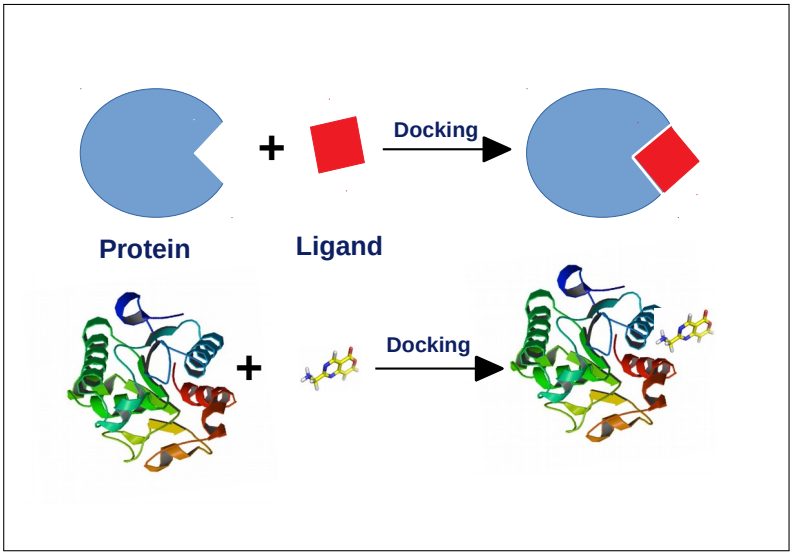

Figure 3.4 – *Illustration of docking process*

3.4 Methodology

Molecular docking methods play an important role in the field of computational chemistry and biomedical engineering (Sousa S.F & M.J 2006). They are frequently used to predict binding orientation and affinity of small molecule to its protein target evaluated by a scoring function (see Figure 3.4). The description of protein-ligand poses in chemical detail provides information on the binding site, bound conformation and binding strength, which is referred to as binding mode (Namasivayam & Günther 2007)(Pagadala *et al.* 2017). In protein-ligand docking, an optimization algorithm (research algorithm) is used to find the best binding mode of a ligand with a target protein by traveling through the search space. The type of the employed algorithm plays therefore a central role in docking accuracy (Guo *et al.* 2014).The scoring function determines the free energy for all given poses of the molecules by modeling their chemistry energy.

Many metaheuristics for searching algorithms have been proposed in the literature. Among these algorithms, we can mention: Genetic Algorithm (GA), Simulated Annealing (SA), and Particle Swarm Optimization (PSO).

Our interest is, however, focused on the BSO algorithm described in section 3.3.6, motivated by its power and success. It is, moreover, efficient, simple to implement, and manages few parameters in comparison with the other metaheuristics. In addition, it has been shown to be effective in several areas such as: data mining (Renli *et al.* 2016), SAT and MAX-W-SAT problems (Sadeg & Drias 2007), information retrieval (Djenouri *et al.* 2018) etc. For those reasons we opted for this metaheuristic (BSO) in our first contribution in this thesis .

In the next section, we will detailed our BSO based metaheuristic solution to solve the molecular docking problem. We named this solution BSO-DOCK.

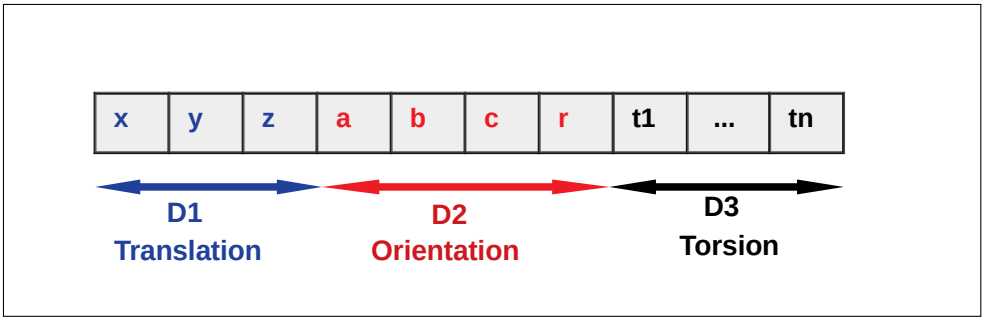

Figure 3.5 – *Representation of the ligand in the Molecular Docking problem*

To design and adapt our BSO metaheuristic to the molecular docking problem we have to define the components of the BSO algorithm: the representation of the problem (encoding solution), the scoring function that evaluates the found solutions), the initial solution *Sref*, the *SearchArea* strategy to find disjoint solutions (diversification of solutions), and the neighborhood search performed by each artificial bee (intensification of solutions or local search) (Saadi *et al.* 2016).

3.4.1 The encoding solution

Our problem can be formulated as an optimization problem (Liu *et al.* 2009), it may be mathematically expressed by a vector **S** and an objective function **E**.

Given: **(S)** = (x,y,z, a,b,c,r, t1, t2,t3 ...tn)

Though: **S***, E(S*) =< E(S)

S: is the vector of decision variables.
E(s): is a scoring function used to predict the free energy of the molecule complex defined by S

We try to find in a very large set of potential solutions S, the best solution S* that minimizes the objective function E(S). The vector **S** can be described as a set of three fields (D1, D2, D3) as shown in Figure 3.5 :

- **D1 (x,y,z)** : Translation: The variables x, y and z represent the position of the center of mass of the ligand.
- **D2 (a,b,c,R)**: Orientation: The orientation is represented by a quaternion, where a, b, c denote the vector of the orientation axis, and **r** denotes the rotation angle around this axis.
- **D3 (t1,t2,t3,...,tn)**: Torsion: ti is an angle associated with the i-th rotatable bond, **n** is the number of rotatable bonds.

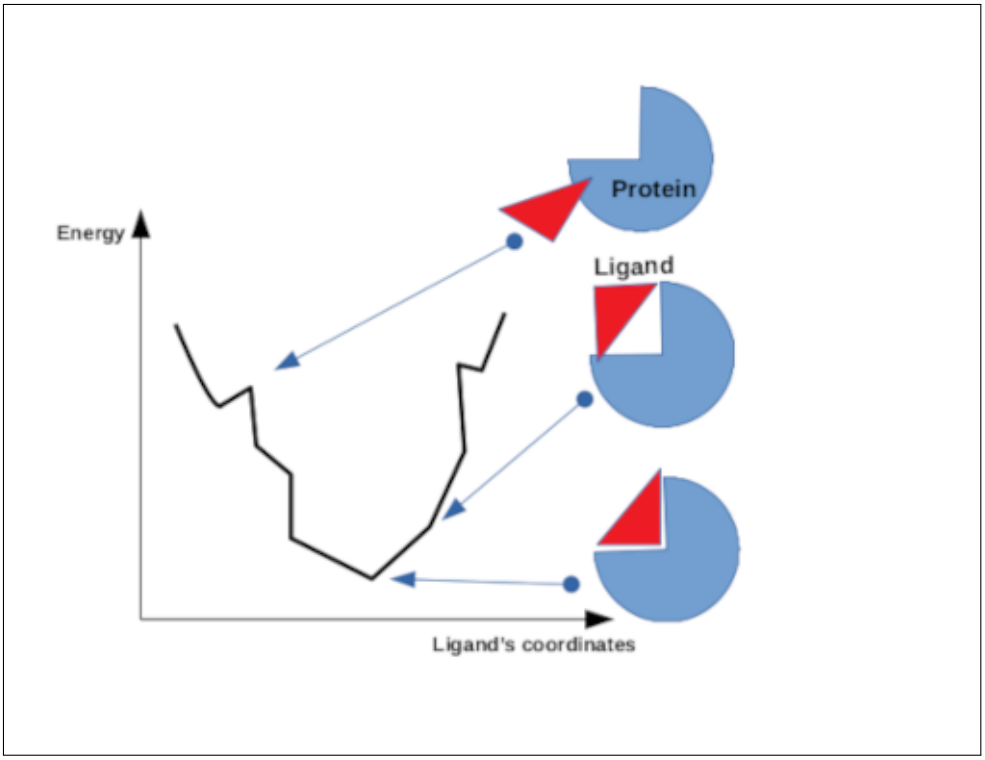

Figure 3.6 – *The Scoring function*

To summarize, we use three parameters to denote the translation, four for the orientation, and **n** parameters for the torsion. Thus, a docked conformation of the ligand will need 7+n parameters to be represented. The **n** parameter can either value 0 for a rigid ligand, or can vary from 1 to 20 or more for a flexible ligand. The complexity of the docking process depends essentially of the **n** parameters.

3.4.2 Scoring function (Fitness function)

The BSO-DOCK scoring function models the affinity between the ligand and the protein, it is based on an energy prediction. The lower is the energy, means the better is the docking (Liu *et al.* 2005) (see Figure 3.6). In our solution the implementation of the scoring function is inspired from the semi-empirical force field scoring function of Autodock4.2 (Forli *et al.* 2012). The latter is the most used and cited framework for molecular docking experiments in the literature. It is also an open source software.

The scoring function involves three energy terms (Liu *et al.* 2009); intramolecular energy of the ligand, intramolecular energy of the protein, and the intermolecular binding energy between the protein and the ligand as shown in Equation 3.1, Equation 3.2, and as illustrated in Figure 3.7.

$$\Delta G = \Delta intra + \Delta inter \tag{3.1}$$

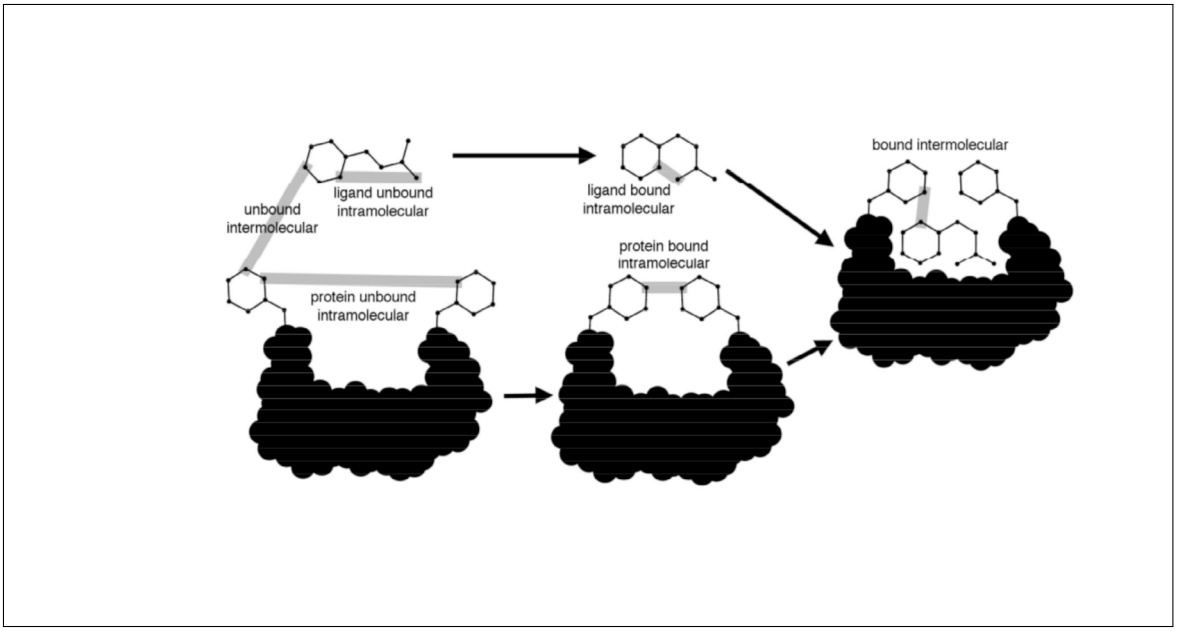

Figure 3.7 – *Autodock Scoring function*

$$\Delta G = (V^{L-L}_{bound} - V^{L-L}_{unbound}) + (V^{P-P}_{bound} - V^{P-P}_{unbound}) + (V^{P-L}_{bound} - V^{P-L}_{unbound}) + \Delta Sconf \tag{3.2}$$

Autodock scoring function calculation involves two steps. In the first step the intramolecular energetics are estimated for both ligand and protein for the transition from unbound states (before interaction) to bound states (after interaction). The second step then evaluates the intermolecular energetics of interaction between the ligand and the protein in their bound conformation (Huey *et al.* 2007).

In Equation 3.2, **L** refers to the "ligand" and **P** refers to the "protein". The first two terms are intramolecular energies for the bound and unbound states of the ligand, and the following two terms are intramolecular energies of the protein. The last term denotes the change in intermolecular energy between the bound and unbound states. And $\Delta Sconf$ estimates the conformational entropy lost during the binding process.

Each of the pair-wise energetic terms in Eq2 includes four types of energy (Scripps-Research-Inst 2009), that is, van der Waals or dispersion/repulsion, hydrogen bonding, electrostatics, and solvation energies, respectively described in equations 3.3, 3.4, 3.5, and 3.6.

$$\Delta GvdW = \Delta G_{vdw} \sum_{ij} A_{ij} \div r^{12}_{ij} - B_{ij} \div r^{6}_{ij} \tag{3.3}$$

$$\Delta Gelect = \Delta G_{elect} \sum_{ij} q_i q_j \div \epsilon r_{ij} r_{ij} + \Delta G_{tor}.N_{tor} + \Delta G_{sol} \sum_{ij} S_i V_j E^{-r^2_{ij}} \div 2.\sigma^2 \tag{3.4}$$

$$\Delta GHbond = \Delta G_{Hbond} \sum i, jE(t)(C_{ij} \div r_{ij}^{12} - D_{ij} \div r_{ij}^{10} + Ebound) \quad (3.5)$$

$$\Delta Gdesolv = \Delta G_{desolv} \sum (i, jS_i.V_j + S_j Vi) e^{r^2_{ij}} \div 2.\sigma^2 \quad (3.6)$$

3.4.3 The Search Area strategy (Regions)

The aim of this step is to generate from a reference solution Sref, a set of **K** disjoint solutions or regions to diversify the solutions, in order to better exploit the search space. Starting from a reference solution Sref (Ds1, Ds2, and Ds3) found by the first bee **Initbee**, a set of **K** disjoint solutions is generated in the search area. For this task, we use a field variable named *Flip*, which is added or subtracted from each variable of each domain to generate another solution from the current solution (or from Sref, the first time). In our case we have three variables *Flip*; *Flip1* used to modify the first domain D1 which models the position of the ligand, *Flip2* modifies the second domain D2 for ligand rotation modulation, *Flip3* modifies the third domain D3 which represents the change of the ligand structure (conformation). The values of the Flip field determine the grid spacing of the search space. All possible k solutions of the search area are generated by successively varying only one field, for example D1, or D 2, or D 3 at a time, and then combine it with the other two unmodified domain D domains.

D1 (x,y,z) and Flip1

Domain D1 takes values between (x-T/2, y-T/2, z-T/2) and (x+T/2, y+T/2, z+T/2), where T is the size the box surrounding the active site. *Flip1* is an integer, with 0 < *Flip1* <= T/n, and **n** is an empirical value. To find regions of D1, we take each variable, for example x, then either add Flip1 to x, or subtract Flip1 from x without changing the other two variables y and z. We do the same operation with the other two variable y, and z to get many solutions or regions.

Example:
We assume that D1 of Sref= (5,2; 35; 10,1)
i. e. x=5,2; y=35; z=10,1; T=60 and n =10, therefore, Flip1=10.

Example of generated solutions:

- D1S1 (**5,2+10**; 35 ; 10,1) = (**15,2**; 35 ; 10,1) The first solution S1 for domain D1
- D1S2 (**5,2-10**; 35 ; 10,1) = (**-5,2**; 35 ; 10,1)
- D1S3 (5,2 ; **35+10** ; 10,1) = (5,2 ; **45** ; 10,1)
- D1S4 (5,2 ; **35-10** ; 10,1) = (5,2 ; **25** ; 10,1)

- D1S5 (5,2 ; 35 ; **10,1+10**) = (5,2 ; 35 ; **20,1**)
- D1D6 (5,2 ; 35 ; **10,1-10**) = (5,2 ; 35 ; **0,1**)

D2 (a,b,c,r) and Flip2

a, b, c coordinates denote the vector of the orientation of the ligand, D2 takes values between (a-T/2, b-T/2, c-T/2) and (a+T/2, b+T/2, c+T/2), that T is the size of the box surrounding the active site, Flip2 is an integer, and 0 < Flip2 < T/n, where **n** is an empirical value too. As we have done with D1, we change one variable by either adding or subtracting Flip2 to/from the variable, while keeping the two other variables unchanged. We do the same operation with the other two variables.

Example:
We assume that D2 of Sref= (-4; 50 ; 15.1)
i. e. a=-4 ; b=50 ; c=15.1; T=80 and n =10, therefore, Flip2=8,
Example of generated solutions for D2:

- D2S1(-4+8; 50 ; 15,1)= (4; 50 ; 15,1)
- D2S2(-4-8; 50 ; 15,1)= (-12; 5O ; 15,1)
- D2S3 (-4; 50+8 ; 15,1)= (-4 ; 58 ; 15,1)
- D2S4 (-4 ; 50-8 ; 15,1)= (-4; 42 ; 15,1)
- D2S5 (-4 ; 50 ; 15,1+8)= (-4 ; 50 ; 23,1)

D3 (t1,t2,t3, . . . ,tn) and Flip3

D3 gives information about rotation inside the ligand known as ligand's conformation, **n** represents the degree of freedom or the number of torsional angle of this ligand. Each ti will take values between (-180, 180). The flip3 between 30 and 60, Flip 3 is an empirical parameter too. The value of Flip3 which is also an empirical parameter is chosen between 30 and 60.

Example:
We suppose Flip3 = 40, n = 5, Sref of D3= (30, 120, 15, -30, 60)
We can generate several solutions from this initial solution by using Flip3 under the condition that (ti+Flip3) is no larger than 180 and (ti-Flip3) is no smaller than -180.
Sref of D3= (30, 120, 15, -30, 60)

- D3S1 = (70, 120, 15, -30, 60)
- D3S2 = (-10, 120, 15, -30, 60)
- .
- D3S9 = (30, 120, 15, -30, 100)
- D3S10 = (30, 120, 15, -30, 20)

Once we have determined the three Flips, we will generate all regions using the three domains D1, D2, and D3, the strategy consisting in varying one domain at a time to generate all possible combinations. For example the possible combinations for D1S1 combining with the two other domain D2 and D3 are:

- K1 = D1S1D2S1D3S1
- K2 = D1S1D2S1D3S2
- K3 = D1S1D2S1D3S3
- K4 = D1S1D2S1D3S4
- K5 = D1S1D2S1D3S5
- K6 = D1S1D2S1D3S6
- K7 = D1S1D2S1D3S7
- K8 = D1S1D2S1D3S8
- K9 = D1S1D2S1D3S9
- K10= D1S1D2S1D3S10
- K11 = D1S1D2S2D3S1
- K12 = D1S1D2S2D3S2
- K13 = D1S1D2S2D3S3
- K14 = D1S1D2S2D3S4
- K15 = D1S1D2S2D3S5
- K16 = D1S1D2S2D3S6
- K17 = D1S1D2S2D3S7
- K18 = D1S1D2S2D3S8
- K19 = D1S1D2S2D3S9
- K20 = D1S1D2S2D3S10

So the value of the first region is: D1S1 D2S1 D3S1
K1= (**15,2**; 35 ; 10,1), (**4**; 50 ; 15,1), (**70**, 120, 15, -30, 60)
We do the same for D1Sx, x varying between 2 and 6 to have all the possible combinations

3.4.4 The neighborhood search

We calculate the neighborhood search space by doing local searches for each domain of the D1, D2, and D3. We take every regions found in the preceding step and do a local search to find neighbors.

Calculating neighborhood for D1(x, y, z)

The neighborhood search is calculated from each solution D1Si found in the previous step, we add then subtract 1 point to each variable in D1S and keep the two other variables unchanged, and then do the same operation with the other two variables. The value of 1 point is equal to 0.375 A, which is the best distance to move a ligand when binding to a protein.

Example:
If the solution is D1S1= (10.2; 35; 10.1) The generated solutions are:

- D1S1V1 = (10.2+0.375; 35; 10.1)
- D1S1V2 = (10.2-0.375; 35; 10, 1)
- D1S1V3 = (10.2; 35-0.375; 10.1)
- .
- .
- D1S1V3 = (10.2; 35; 10.1-0.375)

Calculating the neighborhood for D2 (a,b,c,r)

a,b, and c vary with 1 point .The neighborhood is determined from each D2Si found in the first step by adding or subtracting 1 point for one variable each time. For example, we change b and keep a and c unchanged, then change c and keep a and b unchanged.

Example: If the solution D2S1= (5, 56, 61, 170)

- D2S1V1= (5, 56+ **1 point**, 61, 170) = 5, 56+ **0,375** , 61, 170
- D2S1V2= (5, 56-**1 point**, 61, 170)
- ...
- ...
- D2S1V2= (5, 56, 61-1, 170)

Calculating the neighborhood for D3(T1,T2..Tn)

The neighborhood search for this domain is calculated from each solution D3Si found in the previous step by adding or subtracting alpha angle between 1 and 5 to one variable T_i each time and keep the others variable unchanged.

Example:
If the solution is D2S1= (70, 120, 15, -30, 60), alpha = 5
The generated solutions are:

- D3S1V1= (70+5, 120, 15, -30, 60)
- D3S1V2= (70-5, 120, 15, -30, 60)
- D3S1V3= (70, 120+5, 15, -30, 60)
- ...
- ...
- D3S1V3= (70, 120+5, 15, -30, 60)

The general BSO-DOCK algorithm

Algorithm 6 The general BSO-DOCK algorithm

Begin

```
1: Read empirical parameters N (number of bees), Flip1, Flip2, Flip3,
   MaxIter, t(number of torsion angle)
2: Inputs: So (xo, yo, zo; ) // binding site
3: Output: best solution found Sf(x, y, z; a, b,c, R; n1, n2, n3. . . .nt)
4: Sref = So (xo, yo, zo, ao, bo, co, Ro, T1,T2..Tn) // the initial solution
   generated randomly
5: Fitness(Sref)
6: S=Sref
7: i=0
8: while i < MaxIter do
9:    taboo-list = taboo-list + Sref
10:   solutions = Search-Area(Sref, Flip1, Flip2, Flip3)
11:   table-dance[] = solutions
12:   for each solution S in solutions  do
13:      Fitness(S)
14:      Assign S to eachBee
15:      for each Bee b do do
16:         table-dance[b] = Calc-neighborhood(b)
17:      end for
18:      Sref= the best solution of the table-dance
19:   end for
20:   if (Fitness(Sref) > Fitness(S)) then
21:      S=Sref
22:   end if
23:   i=i+1
24: end while
25: Print S
```

End

3.5 Experiments and results

Simulation set-up

The docking performance of BSO-DOCK was evaluated by comparison with two state-of-the-art algorithms: PSO and GA. The three algorithm are evaluated under the identical conditions:

- The same scoring function is used for all metaheuristics, this scoring function is inspired from Autodock 4.2 semi empirical scoring function (Lopez-Camacho *et al.*).
- Flexible ligands and rigid proteins are used.
- 20 protein-ligand complexes from Autodock package and Jmetal framework are used as benchmark to assess the docking performances and accuracy.
- The 20 complexes of the benchmark are divided into two groups according to the complexity (number of rotational bonds of the ligands).
- The binding site was set with a cubic box (60 60 60), (Angstrom).

$$1 Angstrom = 0,1 nanometre$$

- The maximum number of energy evaluations was set to 200,000.
- Parameters of BSO-DOCK metaheuristic such as the number of bees (n), Flip1, Flip2, and Flip3 were determined empirically.
- JMetalCpp framework is used to implement our BSO-DOCK metaheuristic (López-Camacho *et al.* 2014). JMetalCpp is a software platform, the source code is available, this platform helps designers to develop new metaheuristics for molecular docking problem.

Results and Discussion

The results of the docking simulations obtained with BSO-DOCK, PSO, and GA are shown in Table 3.1 and table 3.2. The docking performance is examined in terms of the lowest energy of interaction. However, the docking accuracy is examined in term of success rate of the docking process by calculating the RMSD (Root Mean Square Deviation) parameter.

The simulation results showed that BSO-DOCK gives good results with the lowest energy results for many ligand-protein complexes, as we can see in table 3.1 and in the most cases BSO-DOCK gives better results than PSO and GA.

For the accuracy, the RMSD (Root-mean square deviation) parameter is used in order to measure the similarity between the ligand before docking and the same ligand after docking. The value of RSMD measures the average distance between atoms of the 2 ligands. If this value calculated after the docking is less than 2 Å, we can say that this docking is acceptable and successful. Equation 3.7 is used to calculate RMSD values (Raschka 2014)

Table 3.1 – *Scoring Function (SF)results*

Complex	Torsion	BSO (SF)	GA (SF)	PSO (SF)
3ptb	0	**-5.91**	-5.15	-5.50
1tnl	2	-6.53	-5.90	**-6.91**
1okl	3	**-7.25**	-6.68	-7.1
1abl	4	-8.64	**-8.80**	-7.25
1abf	4	**-7.89**	-6.92	-7.2
1cbx	5	**-8.55**	-7.25	-8.01
1rls	6	-8.56	-6.25	**-7 .11**
3cpa	7	-9.16	**-9.95**	-8.25
1pph	8	**-8.21**	-7.02	-7.56
1nsd	9	-9.88	-8.16	**-10.41**
1ets	10	**-10.15**	-9.08	-9.25
1phg	12	**-10.90**	-9.20	9.99
6tmn	14	-11.33	-11.02	**-12.02**
2ifb	14	**-11.55**	-10.80	-11.54
1icn	15	**-12.32**	-5.28	-11.52
1apu	16	**-12.45**	-7.65	-12.11
6cpa	17	**-12.91**	-9.25	-11.02
1apt	19	**-13.29**	-8.02	-10.25
1aaq	20	**-13.58**	-7.25	-11.95
1hiv	23	**-14.21**	-9.78	-11.46
Win		14	2	4

$$RMSD(a,b) = 1/n\sqrt{\sum_i (a_i x - b_i x)^2 + (a_i y - b_i y)^2 + (a_i z - b_i z)^2} \quad (3.7)$$

In equation 3.7 a_i and b_i refer to the atoms of molecule 1 and of the molecule 2, respectively. x, y, z are the coordinates of each ligand atom.

Table 3.2 summarizes the RMSD values. It is clear from the results obtained that BSO-DOCK is batter than PSO and GA, indeed, BSO-DOCK is suitable for ligand-protein docking.

Moreover, BSO-DOCK found the best binding results for highly flexible ligands (result from table3.1). These kind of ligands are more difficult for docking than less flexible ligand due to the huge search space to examine. Thus, these results sniffy that BSO-DOCK can be used for more complicated docking type, such as protein-protein docking .

As a conclusion and from the global results, the BSO-DOCK metaheuristic is more suitable for solving the molecular docking problem than the conventional metaheuristics.

Table 3.2 – *Comparison of docking accuracy with RMSD.*

Complex (PDB)	torsion	BSO	GA	PSO
3ptb	0	3.6	1.2	**0.99**
1tnl	2	1.4	5.2	**0.65**
1okl	3	2.2	**1.52**	1.93
1abl	4	3.2	**1.08**	2.09
1abf	4	**1.58**	9.2	6.0
1cbx	5	**0.96**	1.23	1.98
1rls	6	2.14	9.52	**1.58**
3cpa	7	3.1	6.8	**2.00**
1pph	8	**1.64**	1.92	3.1
1nsd	9	8.1	**1.82**	4.05
1ets	10	**1.98**	15.1	5.1
1phg	12	10.2	2.55	**1.58**
6tmn	14	**1.88**	8.2	0.78
2ifb	14	**2.08**	1.65	3.1
1icn	15	**0.97**	10.3	1.11
1apu	16	2.94	**1.50**	4.02
6cpa	17	**1.99**	1.5	3.2
1apt	19	5.2	2.89	**1.24**
1aaq	20	**1.90**	5.81	3.1
1hiv	23	**0.74**	2.0	1.78
Win		10	4	6
Successful		12	9	10

3.6 Conclusion

In this chapter we detailed a new method to explore the search space to get the best position and conformation of a small molecule when it bounds a macromolecule. This solution is based on Bees Swarm Optimization metaheuristic (BSO). We detailed all operators of the BSO-DOCK metaheuristic for application to the molecular docking problem, such as the encoding solution, the fitness function, the determination of the search area, and the neighborhood search. Our solution BSO-DOCK showed a high efficiency compared with the state-of-the-art algorithms. However, the calculation time increases significantly when dealing with very large instances, making it necessary to use parallelization in order to efficiently resolve the MD problem.

4 BSO Metaheuristic on GPU for Molecular Docking problem

4.1 Introduction

As explained in chapter 1, molecular docking methods play an important role in the field of computational chemistry and biomedical engineering (Sousa *et al.* 2006), they are frequently used to predict the binding orientation of a small molecule (ligand) to its protein target in order to predict the affinity and activity measured by a scoring function (Mukesh & Rakesh 2011). The scoring function models chemical interactions and determines the free energy for given poses (Pechan & Feher 2011), it gives information about the stability of the protein-ligand complex. An optimization algorithm (Search algorithm) is used to find the best binding pose of the ligand against a target protein by traveling through the search space. This algorithm plays a central role in determining the docking accuracy (Guo *et al.* 2014). Metaheuristics are widely used as search algorithms in docking methods.

Recent studies have shown; through profiling in docking simulations that most of the CPU time is spent in the evaluation phase (Scoring function) of the metaheuristic (Fang *et al.* 2014). In this phase, the calculation of the scoring function consumes more than 80 % of the total execution time and this is considered as a big challenge in the performance of the metaheuristic. This bottleneck could be resolved through the parallelization of this calculation in order to speedup the docking process.

Recently, Graphics Processing Units (GPUs) have been playing an important role in the general purpose computing field. It exists actually several high level frameworks to program GPUs instead of using low

level GPU APIs such as CUDA. Indeed, although low level APIs can achieve very good performances, it requires high development and maintence costs, it raises some portability issues because developers are required to write a specific code version for each potential target architecture (Hong *et al.* 2010).

In the previous chapter we proposed BSO-DOCK, a bees swarm optimization metaheuristic to solve the molecular docking problem. In this chapter we propose our parallelization of this BSO-based metaheuristic for the molecular docking problem with MapCG framework (Liu *et al.* 2017). This framework in its turn is based on MapReduce (Dean & Ghemawat 2010) programing model which facilitates the task for parallel programming, the aim is to propose a portable application which can run on CPUs or GPUs, or in both CPU and GPUs. This means is that our solution can run sequentially on CPU, or in parallel on GPU without changing the code. In addition, experiments when docking a set of protein-ligand complexes show that our solution achieves a good performance. Indeed, the parallel implementation using MapCG on GPU gains an average speedup of 10x with respect to a single CPU core (Saadi *et al.* 2019a).

The remainder of the chapter is structured as follows. In sections 4.2, 4.3, and 4.4 , we present the background needed to understand the remaining sections. To that purpose, we first briefly introduce the MapReduce model which we have chosen as the parallel model, then we introduce the MapCG framework used to implement this model, and finally we summarize some elements of BSO metaheuristic. In section 4.5 we introduces our GPU parallelization strategies and in section 4.6 we give details on experiments and discuss the obtained results. Finally, in section 5.7 we summarize the results and present some concluding remarks and work perspectives.

4.2 MapReduce

MapReduce is a parallel programming model introduced by Google for large data processing on distributed systems(Dean & Ghemawat 2008). First, this framework was used for the purpose of serving Google's Web page indexing, it substitutes earlier Google's indexing algorithms. Then, it was used for distribute and parallel computing in many fields. In this model, the user defines the computation in terms of a Map and a Reduce functions, and the associate runtime system accordingly parallelizes the computation across different nodes of a cluster or across different CPUs and GPUs cores of the same machine. Developers find the MapReduce model easy to use and beneficial to create parallel programs without any worries about inter-machines communication.

The Map and the Reduce functions are written by the user. The Map allows to divide and distribute the work on different nodes of the cluster and produces a set of intermediate key/value pairs for each data read. The associated library puts together all intermediate values which have

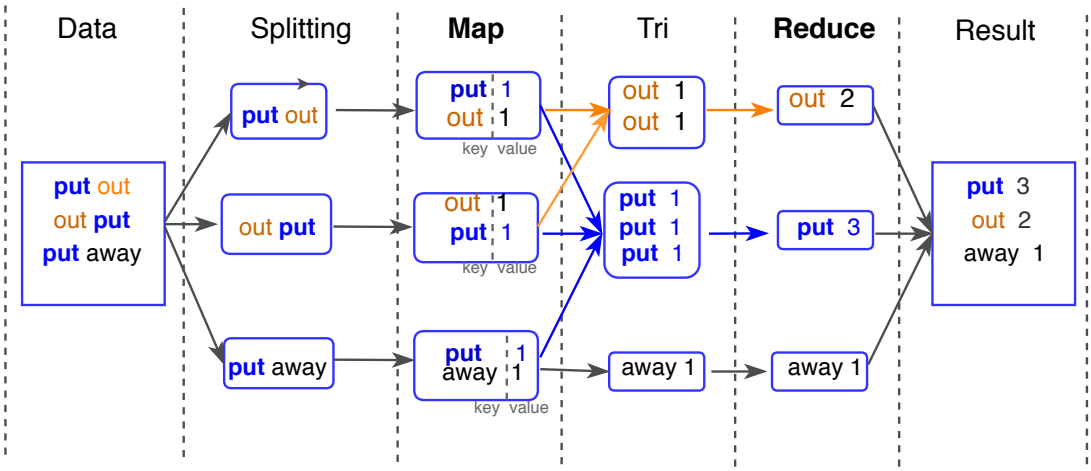

Figure 4.1 – *An example of a text file treated with MapReduce Model*

the same intermediate key *K* and passes them to the reduce function. The reduce function gets the intermediate key provided by the Map function and all values for that key *K*. It incorporates these values together to form results with possibly one output(Dean & Ghemawat 2010).

The steps to follow to write a MapeReduce program are:
1- Choose an approach to split the data so that it can be parallelized by the Map.
2- Choose a good key to use for the problem.
3- Write the program for Map function.
4- Write the program for the Reduce function.

Fig. 4.1 explains an example of a text treated in parallel with the MapReduce Model, the aim is to find the number of occurrences of words in the input text file. First, the lines of the file are splited into blocks. Then in the "Map" phase, keys are created with an associated value. In this example a key is a word and the value is the number 1 to indicate that the word is present once. Then, all the identical keys are grouped together (same words). Finally, and in the Reduce phase, a treatment is performed on all the values of the same key (in this example, the values are added together to obtain the number of occurrences of each word).

4.3 MapCG

MapCG is a framework based on MapReduce model. It allows programmers to write a parallel program that can be executed on CPU and GPU. The programmer only needs to write one version of a program contains essentially the Map and Reduce functions. The framework generates automatically the CPU and GPU versions of the Map and Reduce functions by source code translation, it uses the MapCG runtime library to execute them on CPU or/and GPU, this operation ensures portability with a high level of abstraction (Hong *et al.* 2010).

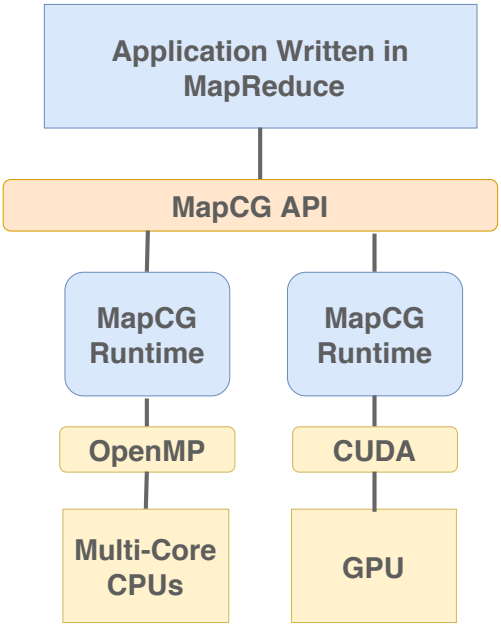

Figure 4.2 – *MapCG architecture overview*

Figure 4.2 shows the MapCG framework architecture. The later contains two parts. The first part, provides the programmers a unified, high level parallel programming environment which allows him to write a MapReduce code once. While the second part represents the MapCG runtime which executes MapReduce code efficiently on different platforms, and bridges the gaps of different hardware features.

In the execution step the input data is split by the *Splitter()* function into pieces, then these pieces are passed to the Map function. The Map function processes the data and emits intermediate pairs (key/value) using MapCG *emit intermediate()* function. The intermediate pairs are then grouped and passed to the Reduce function, which emits data using the MapCG *emit()* function. The data emitted by Reduce can then be obtained by invoking the MapCG *get output()* function.

A hash table is used to group the key/value pair on GPU, it is hard to implement this table on GPU, because the data must be dynamically allocated. For this purpose MapCG uses its one memory allocation system to dynamically allocate memory on GPU and use closed addressing hash table. An other problem is the concurrent insertion issue, MapCG framework uses lock-free algorithm to solve this problem. This algorithm guarantees that the insertion never gets blocked by any particular thread (Hong *et al.* 2012).

4.4 Overview of the BSO metaheuristic

The BSO proposed in (Drias *et al.* 2005) is inspired by the collective bees behavior. It is based on a swarm of artificial bees cooperating together to solve a problem. First, a bee named InitBee settles to find a solution presenting good features. From this first solution called Sref we determine a set of other solutions of the search space by using a certain strategy. This set of solutions is called Search Area. Then, every bee will consider

a solution from the Search Area as its starting point in the search. After accomplishing its search, every bee communicates the best visited solution to all its neighbors through a table named Dance table. One of the solutions stored in this table will become the new reference solution during the next iteration. In order to avoid cycles, the reference solution is stored every time in a taboo list. The reference solution is chosen according to the quality criterion. However, if after a period of time the swarm observes that the solution is not improved, it introduces a criterion of diversification preventing it from being trapped in a local optimum. The diversification criterion consists to select among the solutions stored in taboo list, the most distant one. The algorithm stops when the optimal solution is found or the maximum number of iterations is reached.

4.5 Methodology

In this section, we present the design and implementation of our parallel strategy of BSO metaheuristic. We first describe the overall design with MapReduce model, and then present the implementation with MapCG framwork on the GPU.

All the BSO search algorithm steps such as the determination of the reference solution, the determination of the regions, the neighbor search, all those steps are executed sequentially on the CPU because their complexity is negligible compared to the evaluation step (the scoring function). The later is parallelized using the MapReduce model in which we calculate the free energy on the GPU efficiently by evaluating several solutions in parallel with MAP workers. The main idea is to Maper (divide into several pieces and execute on GPU) the dance table which contains solutions, then chose the best solution within the all solutions found in the first step by the reduce function.

Fig.4.3 explains our parallel approach details. We propose to parallelize the evaluation step of the BSO algorithm. Initially, the dance table generated by the search algorithm is split into several pieces Mi (Maper i) using the split function provided by the MapCG framework. The dance table contains all the solutions generated for one iteration. The number of Mi is determined by this function split function based on the GPU capacity (number of cores) and data size. Each piece Mi is passed to a map function (a map worker) to evaluate the solutions of each piece. During processing, each map function produces intermediate pairs (key, value). The key represents the number of Mi, and the value represents a solution S which contains the position, the orientation and the conformation. We added two elements to the vector of each solution; an index of the solution, and the energy associated for this solution calculated during this step. A solution S is represented as follows:

S = (Index, Translation, Orientation, Rotation, Energy).

These pairs (key, value) are sent by the *intermediate_function* to the reduce function. In this reduction step the *MIN* function is applied to define the best solution with the smallest energy value of each piece Mi using the

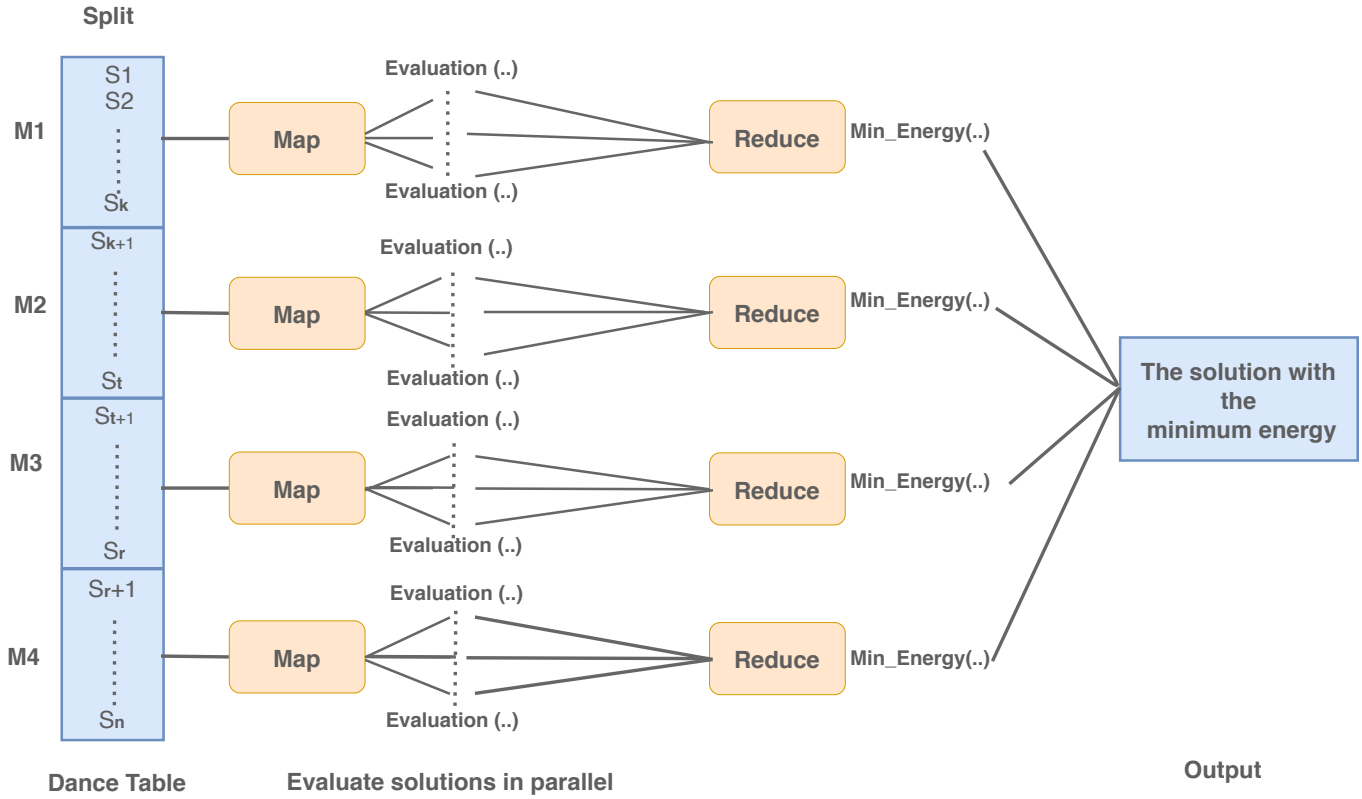

Figure 4.3 – *Parallel BSO design with MapReduce Model*

reduce function. Finally, the pairs generated during the reduction step are emitted by the *emit* function which calls the *get output* function in order to obtain the solution that has the best energy value for all Mi pieces by choosing the min of the different pairs energy.

The pseudo code of the map and reduce functions is given in Algorithm 7:

Algorithm 7 The Map and Reduce pseudo code for the evaluation step BSO

```
1: Map (Key, value)
2: count i [K] // K is the number of Mi pieces
3: for i = 0 to K do
4:     // Mi is a subset of solutions
5:     Evaluate (Si) solutions
6: end for
7: emit_intermediate(...)
8: Reduce (key, value)
9: count i [K]
10: for j = 0 to K do
11:     Choose the solution with minimum energy()
12: end for
13: emit(the best solution)
```

4.6 Experiments and results

This section shows the experimental results. We compare the performance of the BSO algorithm implemented in sequential with the parallel version implemented with MapCG framework.

First, we describe the dataset and the environment used for the evaluation before we show the experimental results. We use GeForce GT 740M card, this GPU is a 1.03 GHz processor with 384 cores and 2 GB of memory size. For the CPU side we use Intel Core i5 model, which is 4 GHz processor with 4 cores. On the software side, we use C++ with MapCG framework, we run this experiments on Linux Mint 17.1 64 bits which was chosen for its stability and performance

The results of this experimentation depend on two factors, the size of the data (large, medium, or small) and the number of iterations of the BSO Metaheuristic. For the size of data we have two elements that increase the complexity and therefore affect the results of our experimentation. First the number of flexible residues on ligand which is known as degree of freedom. Then, the number of atoms in the protein and the ligand. To calculate the energy of interaction between the ligand and the protein, we must calculate the energy between each atom of a ligand with each atom of the protein for n iterations.

The number of solutions generated depends on the size of the vector which represents the solution. The number of variables for the Position and Orientation fields is constant. However, it is variable for the rotation bounds field (flexible residue). So, the number of solutions depends on the number of the flexible bounds in some residues. This latter will vary the number of regions (bees) and the population generated by BSO search algorithm. Table 4.1 describes the population size according to the number of flexible residues bound.

For the benchmark, we use a set of protein-ligand complexes with various size as shown in Table 4.1, we take this datset from RCSB protein data bank (DataBank 2017). The protein-ligand complexes are: COMT, PPAR, ALR2, SRC, ACE, and GPB. For example COMT protein has 3419 atoms , the associate ligand has 52 atoms with 7 rotatable bounds.

In this experiment we vary the number of iteration from 10 to 20 and we calculate the execution time for the sequential and parallel version of BSO, then we calculate the speedup obtained with the parallel version on GPU with respect to the sequential execution on CPU.

Table 4.2 shows that the Parallel BSO algorithm outperforms the sequential algorithm, especially for large data sizes where we evaluate a huge number of solutions. We get better occupancy of the GPU for 20 iterations. We obtained an average speedup of $10x$ with respect to the sequential algorithm. Indeed, the performance of the parallel algorithm with MapCG framework on GPU is better than the sequential algorithm on CPU.

Table 4.1 – *Population size according to data size.*

Data set	nrec	nlig	nrot
GPB	13261	29	1
ALR2	5105	37	6
COMT	3419	52	7
SRC	7158	67	8
PPAR	4430	70	12
ACE	9198	59	13

Table 4.2 – *performance comparison between sequential and parallel BSO and Speed Up.*

Data Size	sequential	parallel	Speed Up
small	140.91	15.51	9.08
medium	550.34	60.21	9.15
large	1236.23	115.51	10.7

4.7 Conclusion

In this article we proposed a new parallel approach for BSO based metaheuristic to solve the molecular docking problem. This approach is based on Map/Reduce Model. We parallelized and implemented the calculation of the scoring function of BSO on GPUs architecture with MapCG Framework. This framework allows us to write one version of the code which can be executed on both CPU and GPU without changing any line of it. The results show a total speed-up exceeding 10x for the scoring function with respect to one CPU version. The MapCG framework and the MapReduce model are a potentially fruitful area for future research in metaheuristic parallelization on GPU and CPUs, and a good tool to accelerate the docking process which can help in drug design speedup process.

5 Parallelization of the Scoring Function for the Blind Docking Problem

5.1 Introduction

Molecular docking (MD) simulates the way that a ligand interacts with a protein target focusing on one binding site. Blind docking is a recent technique which is designed to search the entire surface of the protein to discover new interesting binding sites. Unfortunately, this new docking method is computationally more intensive than MD since its complexity grows exponentially according to the number of binding sites, which severely limits its utilization in practice. This contribution shows a road-map for an efficient parallelization of the calculation of the solvation energy which represents the most time consuming part of the scoring function (Saadi *et al.* 2019b). The latter constitutes a bottleneck in blind docking and in the metaheuristics used to solve this problem. The proposed parallelization approach aims to efficiently exploit the large computing power offered by the latest NVIDIA GPU architectures (Pascal and

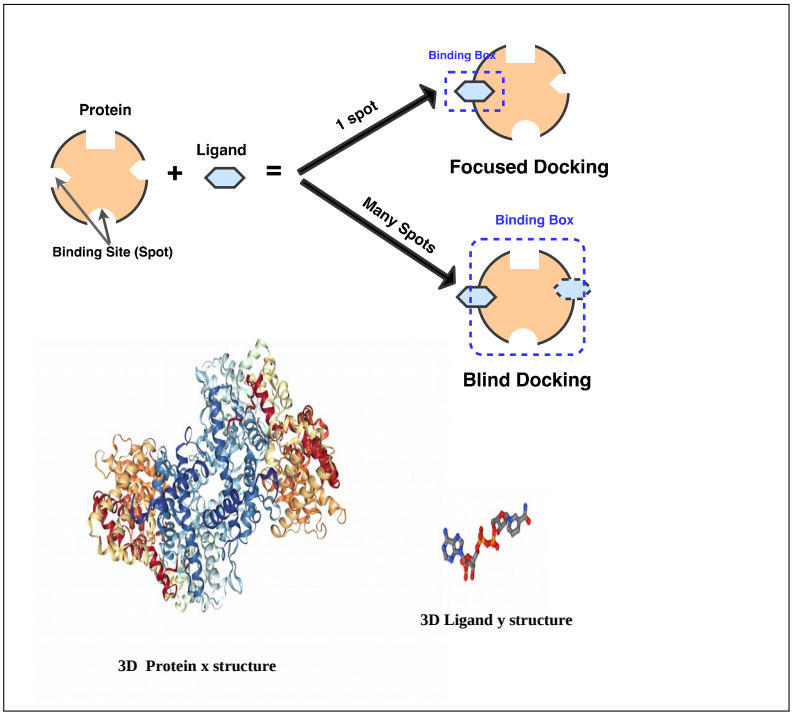

Figure 5.1 – *Illustration of simple and Blind Docking*

Maxwell). Towards this goal, we propose a new parallel approach that exploits several GPU-kernels [1] simultaneously launched by several CPU cores, thereby, speeding-up the computation process and maximizing the GPU utilization.

5.2 The Blind Docking and GPUs

In the last few years, blind docking was proposed to examine the entire surface of target proteins or other macromolecules in order to find new interesting binding sites (Hetenyi & van der Spoel 2006), where small molecules (Ligands) can eventually connect to make a stable complex (as shown in Fig.5.1). This method helps to improve the quality of the docking process, but it exponentially increases the computational time (Hetényi & van der Spoel 2002).

By profiling the docking process, it was noted that most of the CPU time of the metaheuristic used in the docking process is spent in the energy calculation phase (the scoring function), which consumes more than 90 % of the total execution time (Fang *et al.* 2014)(Imbernón *et al.* 2018). Moreover, for blind docking this time is to be multiplied by the number of binding sites which can be large. It is thus of great interest to parallelize this calculation in order to speed up the metaheuristic and the whole docking process.

The scoring function we used is inspired from Autodock, typically it involves four energy terms, the *Van der Waals*, the *hydroen bond*, the *elec-*

[1] units of code executed on the GPU

trostatics, and the *solvation* energies. The calculation of the latter (salvation energy) is the most time consuming.

Of particular interest to us are emerging Graphics Processing Unit (GPUs); initially designed for computer games and display process (Owens *et al.* 2008), but they have recently been successfully used to accelerate many scientific applications (Kirk & Wen-Mei 2016). These graphic cards are nowadays used in many fields ranging from artificial intelligence (Bleiweiss 2008), to smart cars to medical imaging (AUTOMOTIVE 2020)(Science & Imaging 2020), just to name a few. GPUs are also used to accelerate algorithms in the field of bioinformatics, especially for molecular docking (Sukhwani & Herbordt 2009)(Kannan & Ganji 2010). The increase in the use of these devices is supported by their moderate costs and low energy consumption compared to clusters and supercomputers.

In this contribution :

- We first design a CUDA version for the energy calculation to leverage the emergent NVIDIA architectures.
- We then propose a multi-threaded Kernel approach using the Hyper-Q feature (Bradley 2012) in order to increase the performance by maximizing the uses of both CPUs and GPU by simultaneously launching multiple GPU kernels (units of code executed on the GPU).
- To evaluate the performance of our GPU parallel strategy, we first compare its performance with a multi-core CPU version and the sequential counterpart version. We then implement our GPU solution on two different architectures, namely the current (NVIDIA Pascal architecture) and previous (NVIDIA Maxwell architecture) generations. Finally, we experimentally compare our solutions with MURCIA (Zhang *et al.* 2013) (Molecular Unburied Rapid Calculation of Individual Areas) which is one of the best solutions from the literature calculating the solvation term.

The remainder of this chapter is structured as follows. In Section 5.3, we present the background needed to understand the remaining sections. For that purpose, we first briefly introduce the solvation energy term calculation, then introduce the NVIDIA Pascal GPU architecture, and finally summarize some elements of the CUDA programming model. In Section 5.4, we present the related works, such as MURCIA (Zhang *et al.* 2013). In Section 5.5, we introduces our multi-core CPU and GPU parallelization strategies, and in section 5.6 we give details on experiments and we discuss the obtained results. Finally, in Section 5.7 we summarize the results and present some concluding remarks and work perspectives.

5.3 Background

5.3.1 Solvation Term Computation

The following Equation 5.1 explains how to calculate the solvation free energy term ΔG_{Solv} (Huey *et al.* 2007), this equation is based on the semi-empirical force field scoring function of Autodock framework (Morris *et al.* 1998b), the latter is a popular solution composed of a set of tools designed for molecular docking (Morris *et al.* 2009).

$$\Delta G_{Solv} = \Delta W_{Solv} \sum_{ij} (S_i V_j + S_j V_i) e^{-r_{ij}/2\sigma^2} (1) \tag{5.1}$$

In this equation, W_{Solv} is a weight associated with the solvation free energy term (TSRI 2017). S_i, S_j are solvation terms for respectively atom i of the ligand and atom j of the protein. V_i, V_j are atomic volume of atom i of the ligand and atom j of the protein. r_{ij} represents the distance between atom i and atom j. Finally σ is the gaussian distance constant (3.5Å).

The solvation term S for atom i and atom j is reformulated as shown in the following equation

$$S_i = a_i + K|q_i| \tag{5.2}$$

In equation 5.2, a_i is an atomic solvation parameter which depends on the nature of atom i. (e.g it equals to -0.00143 for the Carbon, and -0.00162 for the Nitrogen). K is a constant referred as the charge-based atomic solvation parameter and q_i is a partial atomic charge of atom i.

5.3.2 NVIDIA Pascal Architecture

With Pascal architecture, NVIDIA saves and extends the same programming model (CUDA) afforded by earlier architectures such as Maxwell and Kepler, but it introduces new improvements to performance and power efficiency (NVIDIA 2018) (Whitepaper 2018):

This architecture covers two major variants of cards which can be used for parallel computing: NVIDIA Tesla P100 for data center accelerator and server (Figure 5.2), and Geforce GTX 1080 for Gaming and workstation (Figure 5.3).

The improvements brought by Pascal architecture are :

- NVLink: a new technology developed by NVIDIA to interconnect GPUs in high speed peer-to-peer connection. It is also used to connect CPU to GPUs. It has data rates of at least 80 Giga bytes per second, which is five to 12 times faster than the current PCIExpress. It is also much more energy-efficient than the latter.

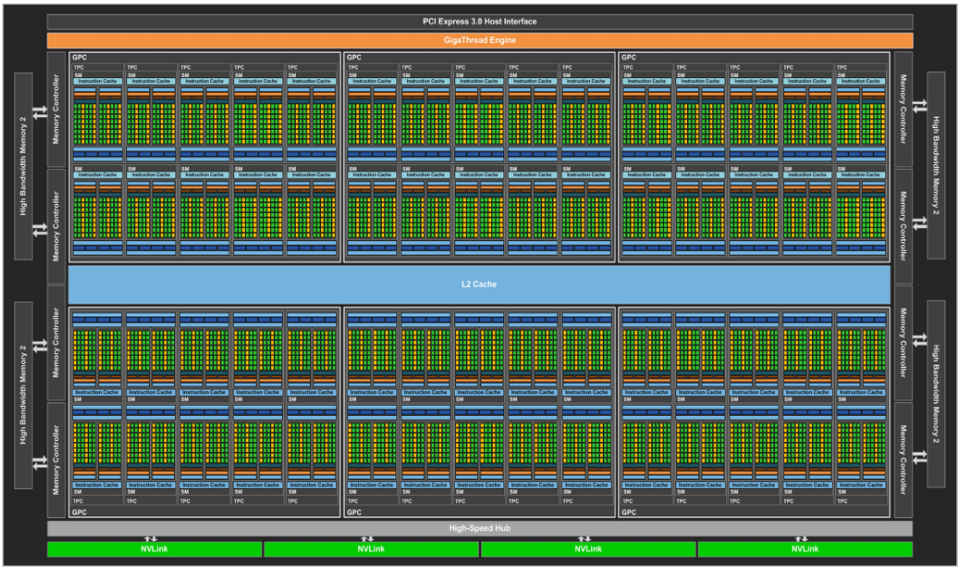

Figure 5.2 – *Pascal Tesla P100 (GP100*

- HBM2 (High Bandwidth Memory 2): a technology which enables multiple layers of memories to be stacked vertically on a single silicon package with the GPU die. It has greater bandwidth, more than twice the capacity and higher energy efficiency than previous architectures. The Pascal Tesla P100 card has a peak bandwidth of 732 GB/s with only 715 MHz memory clock.

- FP16 (colloquially "half precision"): a set of new arithmetic primitives on a half type variable which improves performance up to 2x with less time transfers compared to FP32. It can be used in many fields, especially for Faster Deep Learning.

- Improved shared memory atomic operations 'AtomicAdd()' achieved by implementing native hardware compared to previous architectures were it was implemented in software. It can be used to implement other atomic functions to reduce the overhead in computation. Moreover atomic operations may target the memories of peer GPUs connected through NVLink by using the same API as atomics targeting global memory.

- The unified memory is improved to simplify programming and sharing of memory between CPU and GPU. Two main hardware features enable these improvements: The support for large address spaces and the page faulting capability. With page faulting CUDA does not need to synchronize all managed memory allocations to the GPU before each kernel launche. The needed page is automatically migrated to the GPU memory on-demand.

As you can see in Figure5.3, the GP104 which is used for the GTX 1080 card is equipped with four GPCs (Graphic Processing Clusters), 20 Streaming Multiprocessors (SMs) and 8 memory controllers (=256-bit). Each SM has 128 (CUDA) cores, 256 KB Register File Buffer, 96 KB Shared Memory unit, 48 KB L1-cache and 8 texture units "

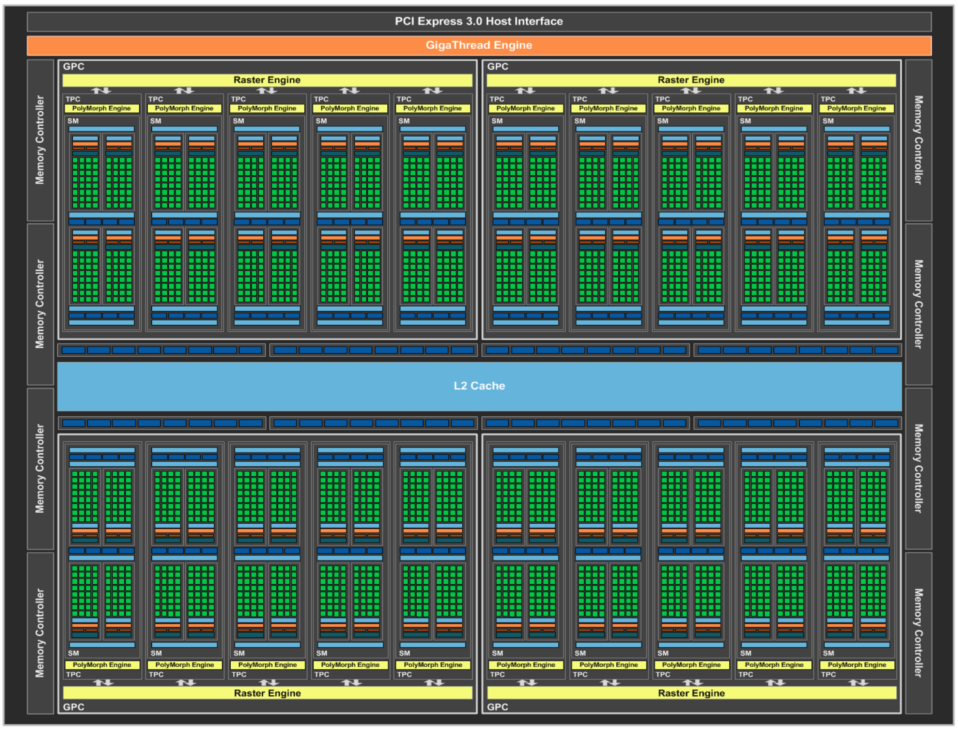

Figure 5.3 – *GTX1080 (GP104)*

5.4 Related work

In this section, we focus on one solution called MURCIA (Molecular Unburied Rapid Calculation of Individual Areas) (Zhang *et al.* 2013). To the best of our knowledge, MURCIA is, so far, the fastest method that can be used to calculate solvation energy on GPUs. For this reason, we chose to compare its performance with the performance of our solution.

MURCIA calculates SASA (Solvent Accessible Surface Area) (Zhang *et al.* 2013), over three functions (GenGrid, Neighbors, and Outpoints) which are described as follows:

1. GenGrid: a grid of points is built to form a sphere around each atom, this procedure uses a grid of 72 points by unit sphere (very low number of points compared to the other methods). However, it guarantees a high precision.

2. Neighbors: for each atom we calculate a list of its closest neighbors, this list is sorted from the closest to the farthest neighbor.

3. Outpoints: this function classifies all grid points generated by GenGrid function in two categories. The first one is called non-buried points, it contributes to SASA and solvation energy term. The second category contains buried points which do not contribute to SASA but give useful information about the molecule surface. This last category can be used in conjunction with the first category for the visualization of molecular surfaces. As shown in Fig.5.4, we take each grid point k of each atom i and calculate the squared distance to the first atom j of the neighbors list. We then test if this distance is smaller than the radius of atom j, and if so, we store point k as

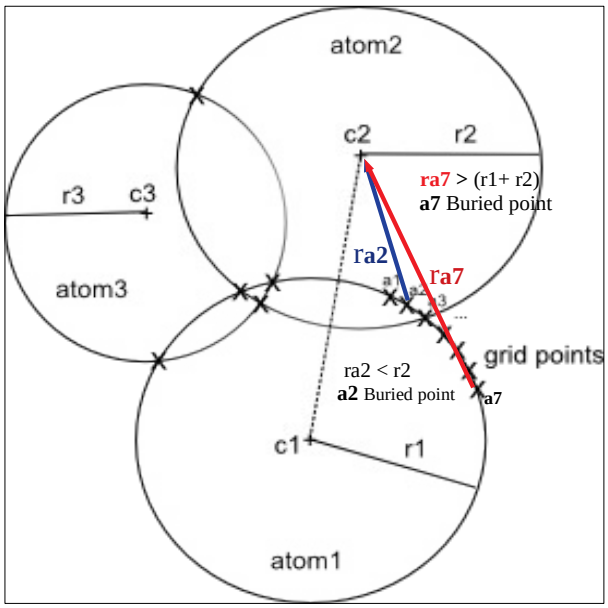

Figure 5.4 – *Illustration in 2D of the SASA calculation in MURCIA*

buried point. Otherwise we flag it as an non-buried point. We do the same procedure by calculating the distances with all other atoms in the neighbors list. Once this procedure is finished for all k points of atom i, we get n contributing points. The individual SASAi for this atom is obtained by the fraction $(n/72)$, where 72 is the number of grid points.

To calculate solvation energy term from SASA, we calculate the sum of all SASAi values by taking in consideration the hydrophobic and hydrophilic nature of each atom i (Eisenberg & McLachlan 1986).

5.4.1 Implementation of MURCIA on GPU with CUDA

Three CUDA kernels are dedicated for the three functions:

1. GenGrid_kernel: each grid point is computed by one thread, the number of threads per block is proportional to the number of grid points per atom (72 points). The total number of calculations for atoms is divided into blocks.

2. Neighbors_kernel: one block is dedicated for each atom with a variable number of threads. Each thread calculates the distance between atom i and all the other atoms j. All threads within a block use shared memory to cooperate and calculate together their neighbors.

3. Out Points_kernel: each thread calculates distances between each grid point of atom i and all of its neighbors j. The number of threads per block is proportional to the number of grid points per atom.

5.5 Methodology

This section introduces the paralelization strategies we applied to solvation energy calculation. First, our sequential and the multicore-based implementations are explained before we provide detailed description of the the CUDA-based parallelization approach.

5.5.1 Sequential Baseline

In this study, we focus on the calculation of the solvation term shown in Algorithm 8. This algorithm calculates the solvation energy for each spot (binding site) *t* by summing the result of *Solvation()* computation for each ligand's atom (*i* loop) with all protein's atoms (*j* loop). We notice that *Solvation()* computation is the implementation of Equation 5.1 (Sec:Background).

Algorithm 8 The sequential pseudo-code

```
1: for t = 0 to nspot do
2:    for i = 0 to nligatoms do
3:       for j = 0 to nprotatoms do
4:          Solvation(ligand[i],protein[j])
5:       end for
6:       solvation[t] += Solvation()
7:    end for
8: end for
```

The key words used in Algorithm 8 are:

- nspot: number of spots (binding sites) on the protein surface
- nprotatoms: number of a protein's atoms
- nligatoms: number of ligand's atoms
- ligand[1 nligatoms], protein[1 nprotatoms]: are vectors that contain the x, y, and z positions and charges of ligand's and protein's atoms.
- Solvation: solvation term calculation for each atom of a ligand.
- solvation: solvation term calculation for each spot

5.5.2 Multicore implementation

The Algorithm 9 shows the Multicore CPU version with OpenMP of the solvation calculation previously explained in Section 5.3. This algorithm calculates solvation energy by computing the solvation energy term for each binding site (spot) *t* on the protein surface (loop 1 in the pseudo-code). The spots are independent of each other, and therefore the solvation

calculation can be executed in parallel. In the multicore version, the loop-parallelization is performed at this level in order to leverage all the cores within the processor. The *solvation [t]* function calculates the solvation for each spot (line 1 and 6 of the pseudo-code) by summing the Solvation term for each ligand's atom with all protein atoms (see loops in lines 2 and 3, and line 6 of the pseudo-code). The latter is implemented using Equation (1) which was previously explained in the Background section.

Algorithm 9 The OpenMP pseudocode of the solvation calculation.

```
0: #pragma omp parallel for
1: for t = 0 to nspot do
2:    for i = 0 to nligatoms do
3:       for j = 0 to nportatoms do
4:          Solvation (ligand[i],protein[j]);
5:       end for
6:       solvation[t] += Solvation();
7:    end for
8: end for
```

As in the sequential code, in this algorithm, *nspot* represents the number of the binding sites on the protein surface, *nportatoms* and *nligatoms* represent the number of the protein's and ligand's atoms respectively. *ligand[1..nligatoms]* and *protein[1..nportatoms]* are vectors that contain the 3D positions of ligand's and protein's atoms.

5.5.3 Implementation on the GPU

Simple kernel implementation

Figure 5.5 represents the the design related to Algorithm 10 on NVIDIA GPU architecture. For each spot *t*, one thread which represents one atom *i* of the ligand, goes through all protein's atom *j* to perform energy. As shown in the device code in Algorithm 10, we reduce the sequential code to just one loop instead of three loops, and use as much as possible threads to perform energy. The total number of threads is equal to the number of spot nspot multiplied by the number of a ligand's atoms nligatoms. We divide this number in blocks, so that we have a good thread distribution in each block. As shown in the host side in Algorithm 10, we choose to transfer all data to the GPU memory and perform energy. This choice is basic, since this wipes unnecessary memory duplicates forward and backward the GPU and host memories. After finishing the calculation, we transfer the result to the host memory.

In Algorithm 10:

- each thread *t* which represents one ligand's atom *i*, performs the energy with the entire protein atoms by executing the *Solvation* computation within the *j* for loop.

Algorithm 10 The parallel pseudo code

Host side

1: Copy Data CPU to GPU

- protein vector : protein [1.. nprotatoms]
- ligand vector: ligand[1 .. nligatoms]
- autodock parameter vector : parmvect(...)
- nligatom, nprotatom, nspot

2: Call GPU Kernel

- SolvKernel(nligatom, nprotatom, nspot, ligand[],protein [],parmvect (...))

3: Copy Energy result GPU to CPU

Device side

```
1: __global__ SolvKernel(nligatom, nprotatom, nspot, ligand[],protein
   [],parmvect (...))
2: total = nspot * nligatoms
3: t = blockIdx.x * blockDim.x + threadIdx.x
4: idSpot = t / nligatom
5: if t <total  then
6:    i = t % nligatoms
7:    for j= 0 to nprotatoms  do
8:       Solvation(ligand[i],protein[j])
9:    end for
10:   AtomicAdd( solvation[idSpot],Solvation)
11: end if
```

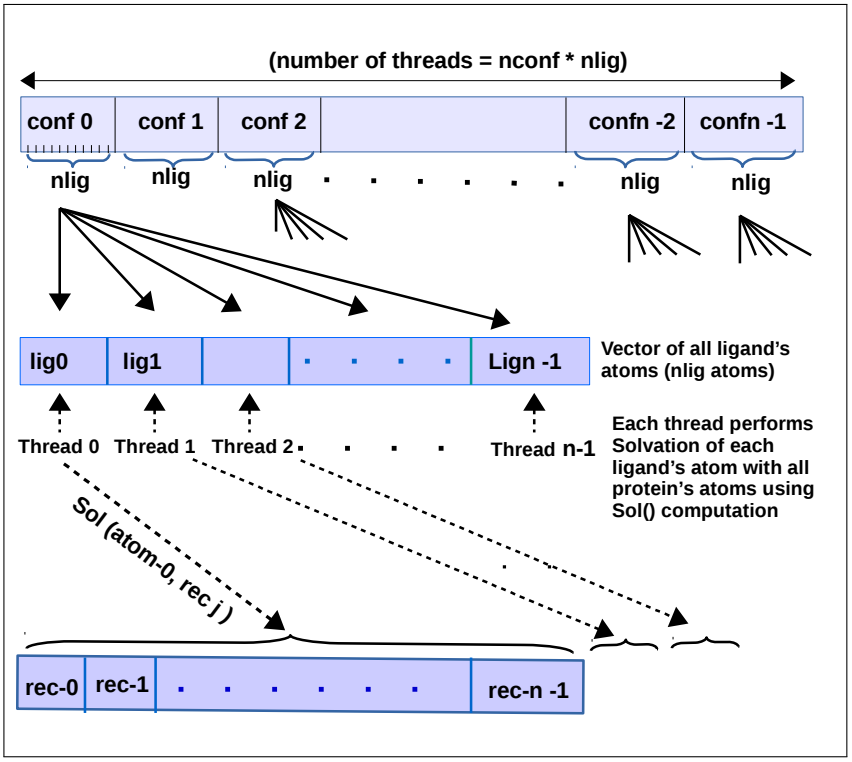

Figure 5.5 – *CUDA design (conf i represents the binding site i or spot i on the protein surface*

- *parmvect()* is a vector which contains all the constants necessary to calculate the solvation energy.
- Using CUDA *atomicAdd* function to Calculate solvation for each spot by summing *Solvation* result for each ligand's atom within the same spot.
- having as many threads as *nligatoms*nspo* (namely *total*).
- having as many thread blocks as *total* divided by the number of threads within a block.

Optimized kernel implementation

Getting data from GPU's global memory is critical to the performance of a CUDA kernel. In order to optimize kernel in Algorithm 10, we decided to use the tiling technique and shared memory to efficiently move data. A structure of arrays is used for the representation of the protein and ligands molecule to guarantee the accesses to the device memory in a coalesced way. To optimize our kernel, we first used shared memory to save x,y, and z coordinates of the ligand atoms loaded by one thread and shared with all threads within the same block. We then used the tiling technique to group the small number of protein atoms in tiles. Threads in the same block collaborate to move this group from global memory in a single access and put it in the shared memory. During the process of energy calculation each thread which represents one ligand atom performs energy with tiles instead of working with just one protein atom, and brings

its data from global memory in a single way. By using this technique we have considerably increased the performance of the kernel of algorithm 10.

5.5.4 Implementation on the GPU with CUDA Multi-Kernel

In this work, we design a CUDA with Multi-kernel technique for solvation energy calculation on the NVIDIA Pascal architecture. It is an extension of the design we proposed in our previous section. In order to maximize GPU utilization, we propose an approach to launch several GPU-kernels simultaneously by using the Hyper-Q technique (Bradley 2012). This technique enables multiple CPU threads to launch simultaneously many instances of our kernel to calculate solvation energy on the GPU. HyperQ permits multiple CPU threads to execute a code on a single GPU concurrently. It also allows the CPU to launch up to 32 simultaneous tasks on a GPU, which increases the entire number of connections between the CPU and the device (GPU). This is different from the single connection when we used the GPU without HyperQ (suach as Nvidia Fermi architecture). Therefore, this hybrid hardware and software solution significantly reduces CPU idle times while increasing GPU utilization. Fig.5.6 explains our new CUDA design. First, m CPU threads (streams) launch simultaneously m instances of our *kernel* to calculate solvation energy on the GPU. Each *kernel* calculates solvation energy for a group of binding sites (spots). The number of launched kernels depends on the hardware features, (i.e. how many CPU and GPU cores are available on the system) and the size of the problem (i.e. small, medium, or large). The number of binding sites treated by each kernel is equal to the total number of binding sites (nspots) divided by the number of launched CPU threads (m). After launching kernels, each thread on GPU represents one atom i of the ligand, and goes through all the protein's atoms j to perform solvation energy calculation. This operation is repeated for all spots. The total number of threads by kernel is equal to the number of spots by stream (CPU) multiplied by the number of a ligand's atoms *nligatoms*, and the total number of threads on the GPU is nspot multiplied by *nligatoms*.

As Algorithm 11 shows, on the host side we initialize and read the proteins and ligands atoms 3D coordinates (x,y,z). We read also the vector *parmvect()* that contains all the constants necessary to calculate solvation energy. Then, we read the number of atoms in ligand and protein and the number of CPU threads (nstreams) and the number of binding sites *nspot*. After reading all necessary data, the code calculates the number of spot to be treated by each GPU Kernel (line 2 in the Algorithm 11, then it creates CPU threads (streams) which call simultaneously multiple instances of the Kernel (*SolvationKernel*) on GPU.

On the device side, each instance of the kernel calculates solvation energy for a group of spots. The GPU thread represents one atom i of a ligand and calculates the solvation energy with the entire proteins atoms by executing the *Solvation* function within the innermost loop. In that loop ligand[i] and protein[j] are elements of arrays of vectors of three dimen-

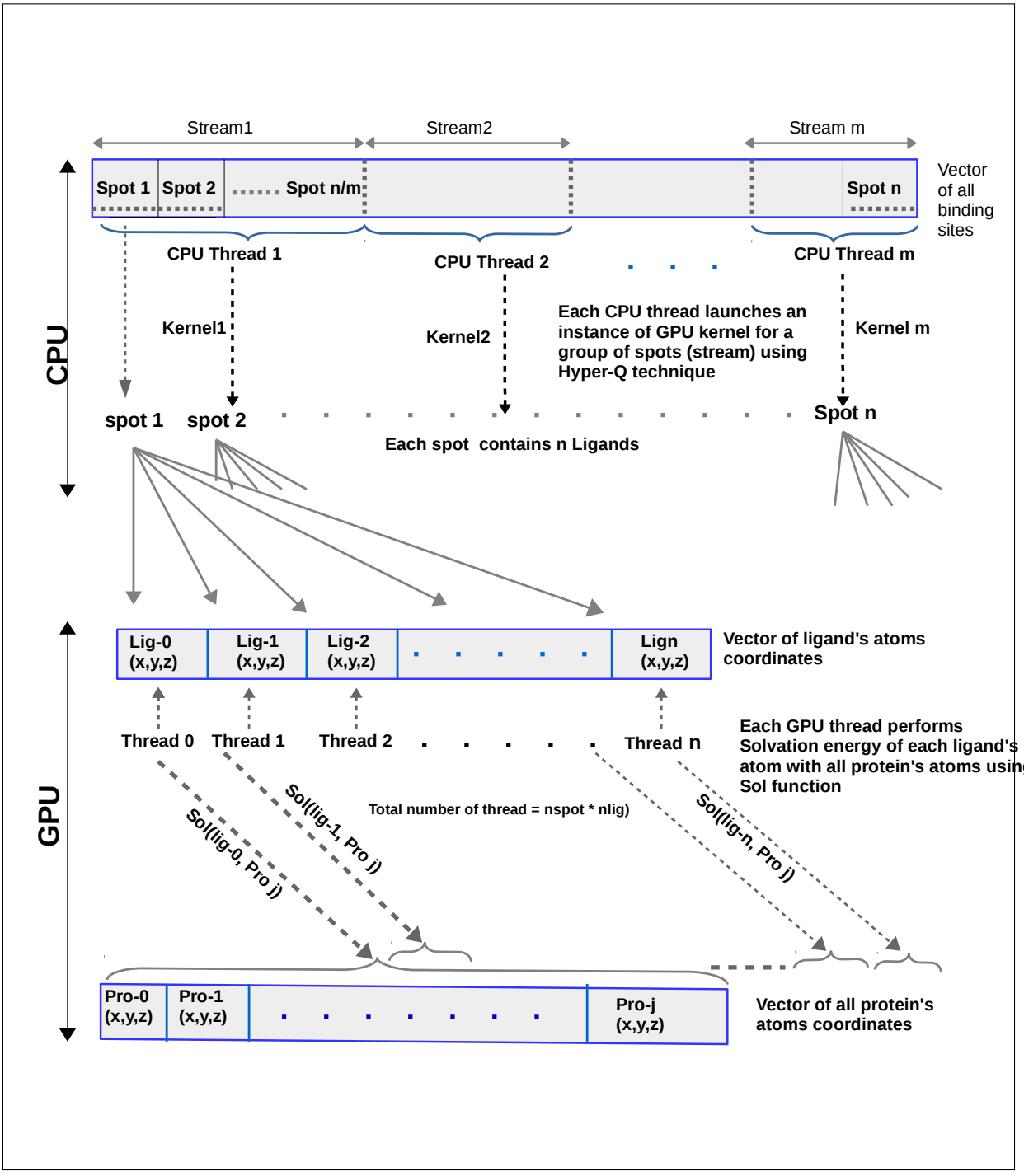

Figure 5.6 – *CUDA design with Hyper-Q technique*

Algorithm 11 The CUDA pseudocode of solvation calculation

Host side

Read data

- nligatoms,nportatoms,nspot,nstream
- protein[1..nportatoms],ligand[1..nligatoms]
- parmvect(...)

Call GPU Kernel instances using HyperQ

```
mspot = nspot / nstreams
for i= 0 to nstreams do
    SolvationKernel (streams[i])
end for
```

Device side

```
__global__ SolvationKernel(nligatoms, nprotatoms, nspot, protein[],ligand[], parmvect(...))
total = mspot * nligatoms
id = blockIdx.x * blockDim.x + threadIdx.x
idSpot = id / nligatoms
if id < total then
    i= id % nligatoms
    for j= 0 to nprotatoms do
        Solvation(ligand[i],protein[j])
    end for
    AtomicAdd(solvation[idSpot],Solvation)
end if
```

sions, x, y and z. The numbers 'i' and 'j' represent indexes in the arrays ligand and protein. For example: ligand [3]=(3;5.2;6) is the third atom of the ligand which has three coordinates. protein [2] = (5;-2;1) is the second atom of the protein which has three coordinates.

The *AtomicAdd* function (line 10 in the Algorithm 11), which is enhanced in Pascal architectures, is used to calculate solvation energy for each binding site (spot) by summing *Solvation* result for each ligands atom calculated in the previous step.

To enhance the performance of our solution, we used the following techniques:

- We used Tiling technique with shared memory to group a small number of the protein atoms equal to *warp* size (32 threads). Tiles are moved from global memory to shared memory in a single access. Hence, during the process of energy calculation, each thread (one atom of the ligand) performs solvation calculation with tiles (32 atoms) instead of working with just one protein atoms. This technique ensures the accesses to the global memory in a coalesced way.
- Minimization of redundant accesses to global memory by using vectors structures for ligand and protein data which avoids threads divergence within the same warp, and ensures the access to the global memory in coalesced way.
- We used the unified memory to benefit from the fault pagination offered by Pascal architectures, where the needed page (data) is automatically migrated to the GPU memory on-demand, and with the very high data transfer rate offered by the Nvlink interconnections and the HB2 memory technology.

5.6 Experimental Results

In this section we show the performance of our GPU-based solution described in the previous section and compare it with the performance of the sequential and the parallel multi-core CPU implementations.

We start by describing the benchmark and the environment used for the evaluation, and then present the experimental results. In this experimentation, we targeted one of the latest GPU architectures (Pascal). We specifically used one of the latest GPU cards of this architecture family, namely the GTX 1080 Ti card. The latter uses a 1480 MHz processor with 3584 cores and 11 GB of memory (See Table 5.2). For the CPU side as shown in Table 5.3, we used two Intel Xeon E5-2650 processors, equipped with a 2.20 GHz processor with a total of 24 cores (48 CPU threads) and 128 GB of RAM. On the software side, Open Multi-Processing (OpenMp) API is used for multi threading computation on CPUs (Board 2018), and CUDA platform is used for the GPUs programming. More precisely, we

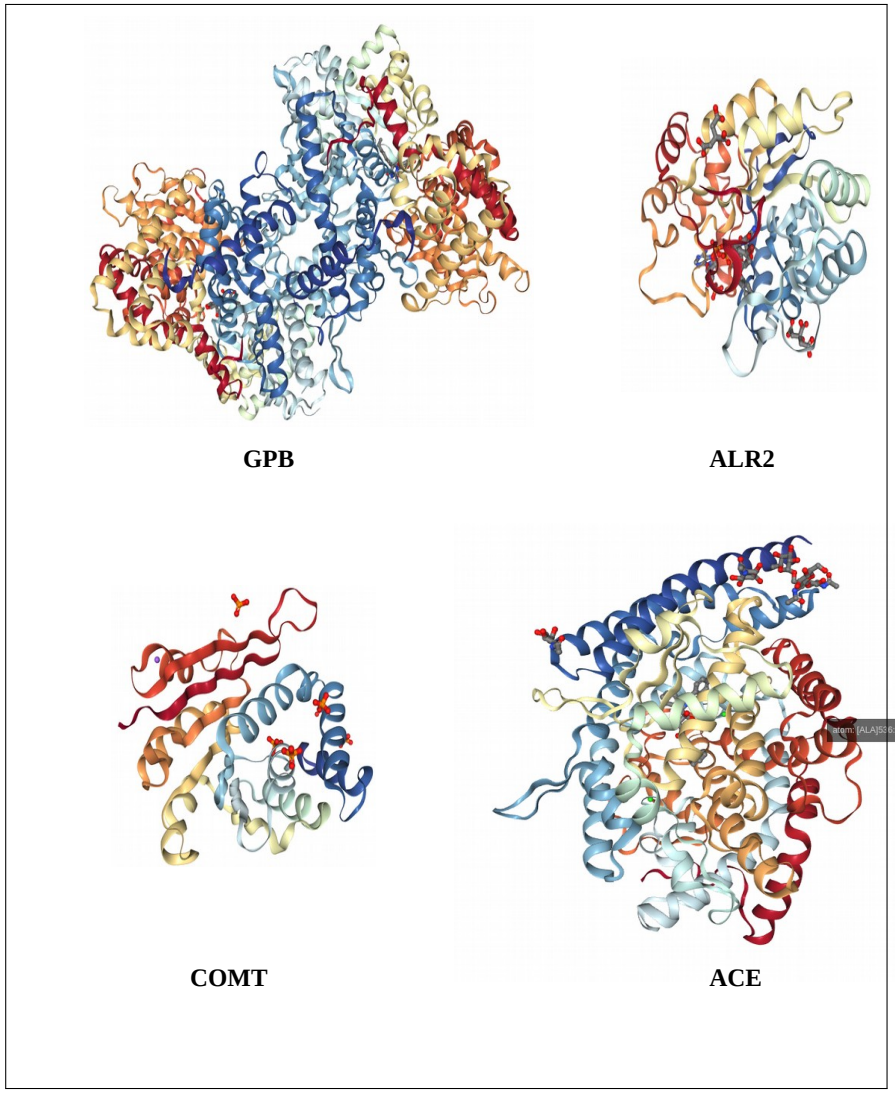

Figure 5.7 – *Protein-Ligand dataset 3D structure*

used the CUDA 8.0 platform that leverages the latest Nvidia Pascal architecture.

For benchmarking, we used the Useful Decoys Database (DUD), a dataset designed for virtual screening and docking tests (Huang *et al.* 2006). We chose ligand-protein compounds with various sizes that require different amounts of resources in the docking process. As shown in Table 5.1, the protein in the first compound named GPB has 13261 atoms, and the associate ligand has 29 atoms. The second ALR2 has 5105 atoms for the protein and 37 atoms for the ligand.

Table 5.1 – *Protein-Ligand pairs characteristics*

Data set	nportatoms	nligatoms
Comp 1 / GPB	13261	29
Comp 2 / ALR2	5105	37
Comp 3 / ACG	9198	59
Comp 4 / COMT	3419	52

In our experiments, we perform multiple evaluations and comparisons:

1. We compare the performances of our solution implemented on GPU with the Multi-core CPUs implementation and the sequential code. Then, we study the influence of the number of binding sites on the performance of the docking process. To this end, we simulate a simple molecular docking by calculating solvation term for one binding site (one spot), then we increase the number of spots to deal with the blind docking and explore the whole surface of the protein (subsection 5.6.1).

2. We compare the implementations of our parallel solution on Pascal architecture and on the previous Nvidia architecture (Maxwell) (subsection 5.6.2).

3. We compare our parallel strategy with the best solution of the literature which is MURCIA (results in subsection 5.6.3).

5.6.1 Performance comparison between the GPU and the multi-cores CPU implementation

Our experiments start with the simulation of a simple docking with one binding site. We run the sequential code on just one core, then we execute the parallel code to calculate solvation energy on multi-core CPUs (24 cores) using OpenMP. Finally, we run the multi-threaded CUDA code on GPU (Pascal GTX 1080 Ti card). For the time consumption (see Fig.5.9), we observe that the performance on GPU is better than on one CPU and on multi-core CPUs. However, the difference in not very large. Also, the performance on one CPU is a little bit better than the performance on multi-core CPUs for all compounds of the dataset. This is due primarily to parallelization overhead with the OpenMP API, and secondly to the fact that we do not have enough tasks to exploit CPU cores (24 cores).

For the speedup, the solvation term calculation got an average acceleration between 2.5x and 4.20x on the GPU version with respect to the sequential code. This depends on the protein and ligand size. We can explain this result by the fact that we do not have enough threads to occupy all the GPU blocks. This means that we did not fully exploit the computation power of the GPU. For the multicore CPU version, we did not get any acceleration due to parallelization overhead with the OpenMP API. Hence, for simple docking, GPU is better than both parallel multicore CPU and sequential code.

To fairly compare the parallel GPU solvation calculation with the parallel multi-core CPU implementation for blind docking, we ran the multicore code with different number of CPU threads (1, 4, 8, 16, 32, 48, 64, and 96) for a large benchmark with 200 spots and the largest protein-ligand compound (Comp 2 in Table 5.1). The goal is to get the best configuration for the multicore CPU (optimal number of threads) in a 24-cores CPUs server. As shown in Fig.5.8 , the best execution time was reached with 48 threads. We thus used the latter configuration for the multicore CPU implementation.

In the second step of our experiments, we increased the number of binding sites (spots) to simulate blind docking. We study the influence of this parameter on the performance of our parallelization strategy. In our tests, we used 10, 20, 50, 100, 200 and 300 binding sites.

In terms of accuracy, solvation calculation for blind docking with the parallel version gives the same values as the sequential code.

We notice that increasing the number of spots has a significant influence over the performance of the three approaches. Escalating it greatly increases the computation time in the case of the sequential code. It also increases the computation time for the multi-core approach in smaller proportions compared to the sequential version. The GPU version, however, gives relatively good results, with smaller increase in computation time. For example, when using 300 spots the average execution time is 10544 ms for the sequential code, 200 ms for Multi-Cores CPU, and just 33 ms for GPU. As can be seen from Fig.5.9 , with GPU the net speedup increases exponentially with the number of binding site (spot). e.g., for the first compound with 1 spot it equals 4.20x, 27.50x for 10 spots, 48.98x for 20 spots, 110.82x for 50 spots, and 213.39x for 100 spots. For the multi-core CPU version, the speedup increases exponentially with the number of spots from 1 to 100. Then, it becomes stable (15x-16x) when the number of spots varies from 100 to 300. From these results we can observe that the parallelization is far better with GPU than with multicore CPU. We thus conclude that the use of GPUs to parallelize blind docking is very promising.

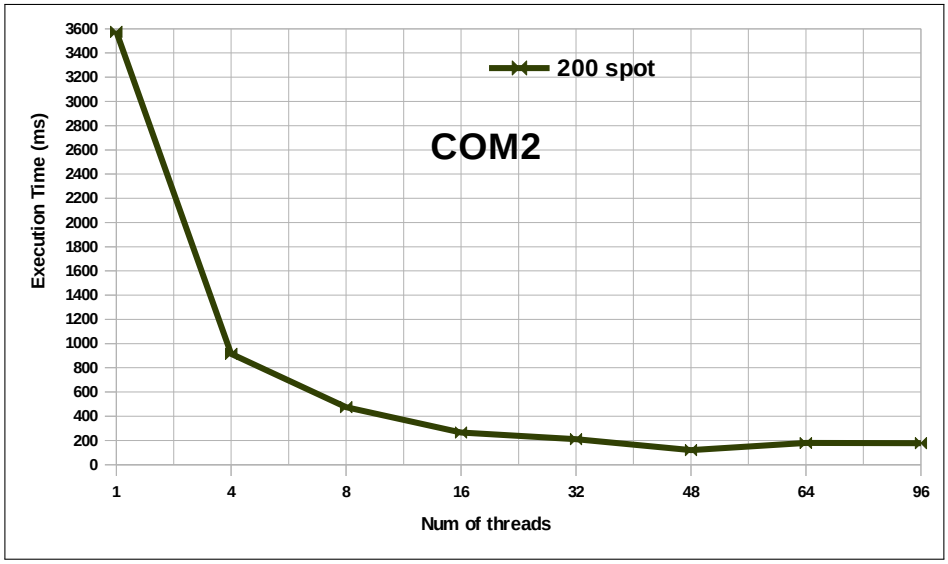

Figure 5.8 – *Multicore performance with 1, 4, 8, 16 , 32 , 48, 96 threads*

5.6.2 Performance comparison between Pascal and Maxwell implementations

In this section we compare the results obtained with the Pascal architecture with the ones obtained with the previous NVIDIA architecture (Maxwell). We implemented our code on Titan X card which represents the Maxwell architecture. We evaluated the computational performance of our parallelization strategies on this architecture and compare it with the Pascal implementation. We used the same dataset for the benchmarking (see Table 5.1). Table 5.2 details the features and shows a hardware comparison between the two GPU cards which represent the two architectures. Fig.5.10 shows time consumption for both architectures, and the speedup obtained with these two cards with respect to sequential code. The experimental results show that Maxwell gives good results, for 100 binding sites it achieves an average 121x speed-up with respect to the sequential code. This speed-up enhances exponentially with the number of spots. However, as we can see on time consumption and speedup graphs and for the entire dataset, Pascal consumes less time to calculate solvation energy for the scoring function and accelerates better the sequential code. We can conclude that Pascal-based solvation term outperforms Maxwell-based solvation term calculation; using the Pascal architecture is very beneficial for blind docking with lower cost and better power consumption.

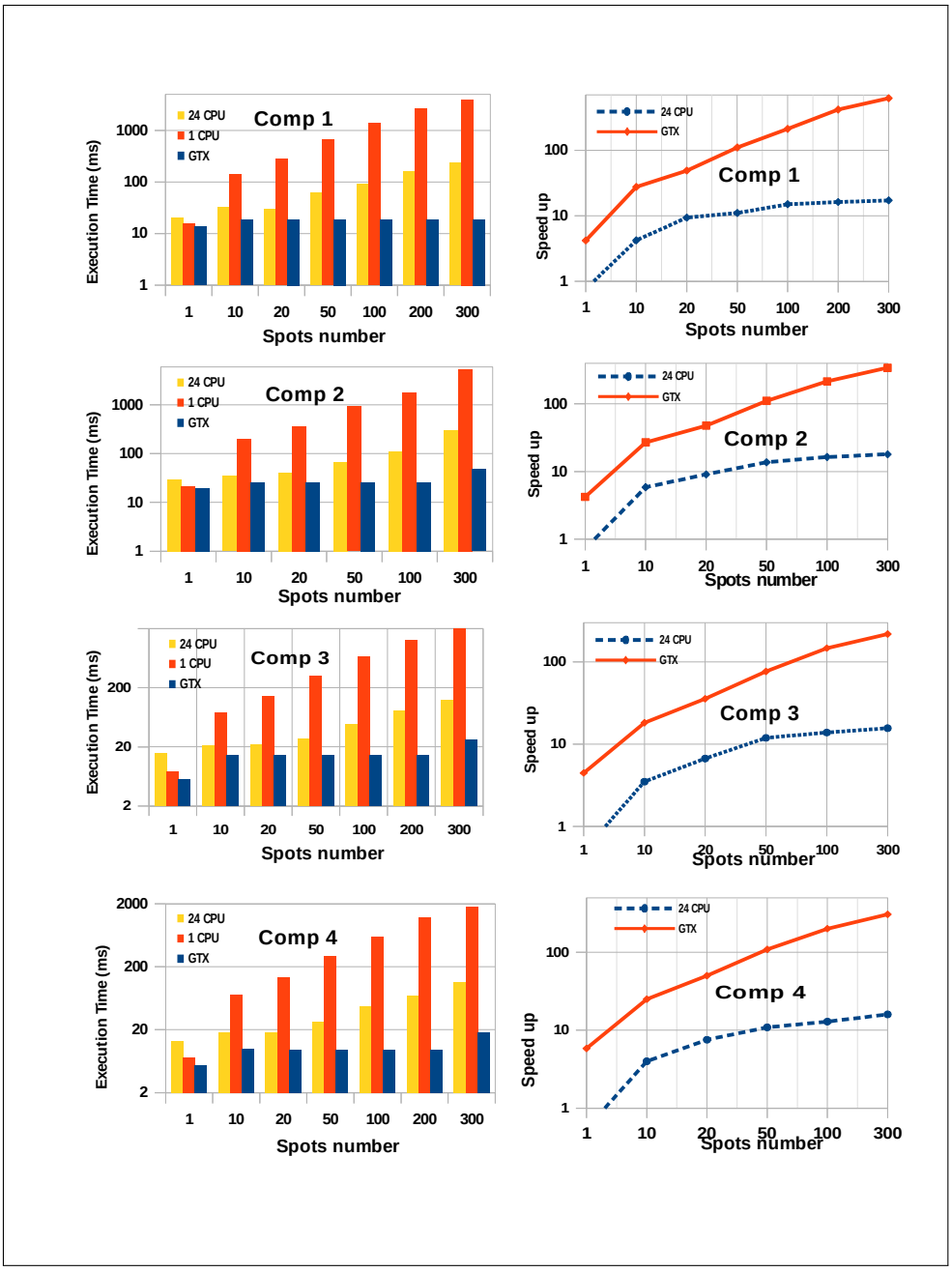

Figure 5.9 – *Execution time of the sequential version (1-core CPU), and the parallel version (24-core with 48-threads CPU) and GPU GTX Pascal card (The left side). Speedup obtained with Multi-core CPU and GPU (Pascal) with respect to one-core CPU (The right side). NB:Y axis is in log scale).*

Table 5.2 – *GPUs hardware resources comparison*

GPU name	TITAN X (Maxwell)	GTX 1080Ti (Pascal)
Price (Amazon)	1400 $	700 $
Year of release	2016	2017
Raw computational power		
number of multiprocessors	24	28
Number of cores	3072	3584
Cores frequency	1417 MHz	1480 MHz
Peak processing	10.16 TFLOPS	10.6 TFLOPS
Compute capability	5.2	6.1
Memory		
Global memory	12 GB	11 GB
Shared memory	49 KB	49 KB
L2 cache	3.1 MB	2.8 MB

Table 5.3 – *CPUs Hardware resources*

Feature	Intel CPU
Model	E5-2650 v4
Year released	2016
Raw computational power	
Number of cores	24
Number of threads	48
Cores frequency (MHz)	2,20 GHz
Peak processing (GFLOPS)	254
Memory	
Size (GB.)	128 GB
Cache	
L1/Shared memory.	32 KB
L2 cache	256 KB
L3 cache	32 MB

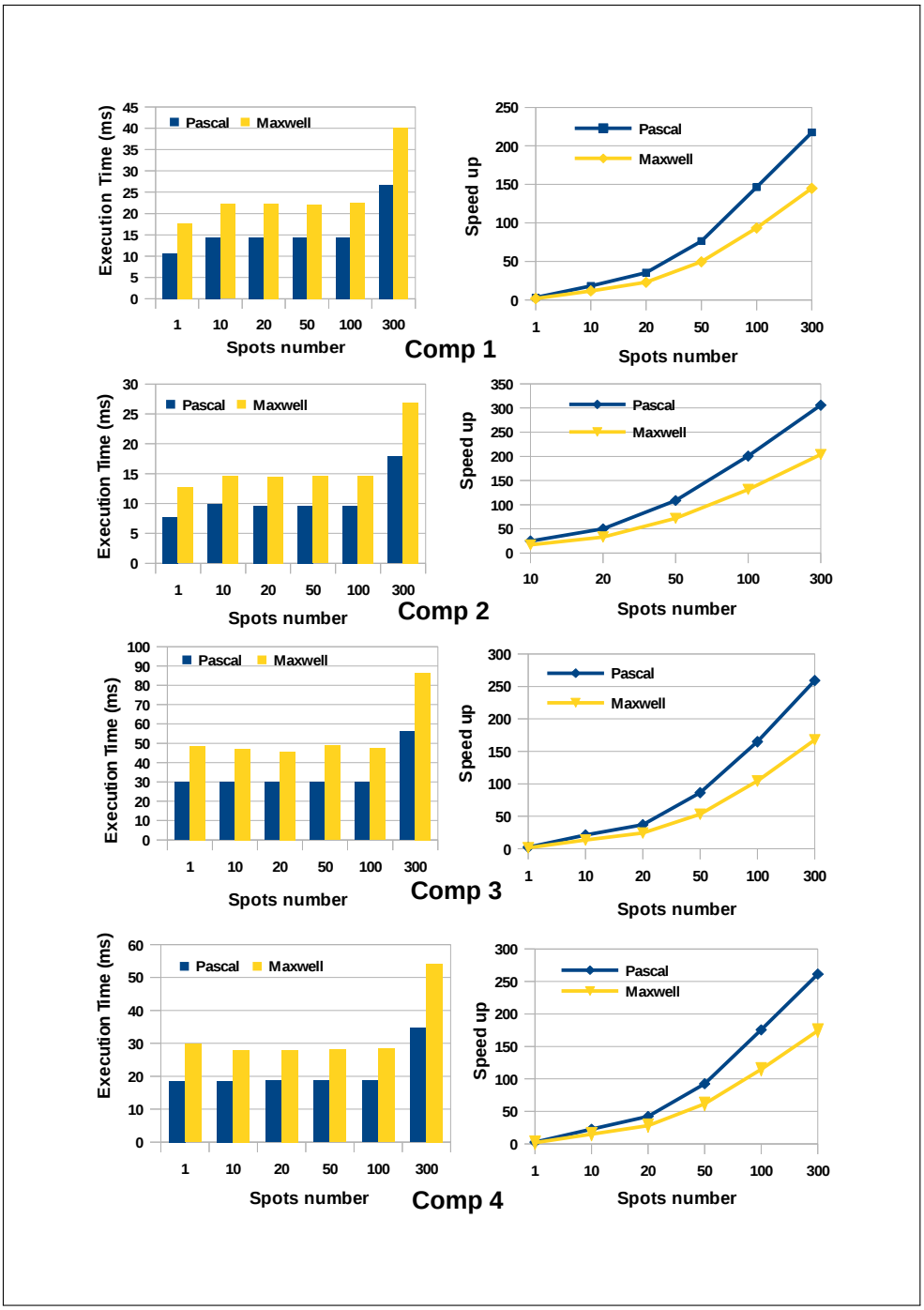

Figure 5.10 – *Performance comparison between PASCAL architecture (GTX 1080 Ti card) and Maxwell architecture (Titan X graphic card),*

Table 5.4 – *Performance comparison with MURCIA*

Data set	Our solution ms	MURCIA K1 ms	MURCIA K2 ms
CP1	13.11	111.78	104.80
CP2	20.80	69.29	67.80
CP3	12.23	26.91	25.95
CP4	9.46	40.73	35.91

5.6.3 Performance comparison with the state of art (MURCIA)

In this section we compare the performance of our solution with the performance of MURCIA (see Background section) which is considered as the fastest method to date for the calculation of the solvation energy term on GPU architectures. For the purpose of this experiment, we have executed two versions of MURCIA on the same GPU architecture which we have used to execute our solution. The first version (called MURCIA-K1) is developed with CUDA, whereas the second version (MURCIA-K2) uses CUDA with Trust library (NVIDIA 2017) and Particle systems model (Green 2010) to enhance the performance of MURCIA K1. We compare the results of the implementations of the two versions of MURCIA with the results of our parallel strategy.

Table 5.4 shows the results of this comparison. We found that our solution outperforms both MURCIA versions i.e. runs faster then MURCIA K1 and MURCIA K2. If we take the first compound from the dataset, the execution time for MURCIA K1 and MURCIA K2 are 111.78 ms and 104.80 ms respectively, however it takes just 13.11 ms with our solution.

5.7 Conclusion

In this chapter, we showed our contributions to improve the performance of the blind docking problem by parallelzing efficiently the calculation of the solvation energy term, which is considered as the most time-consuming part of the scoring function of the meta-heuristic used to solve the blind docking problem. Towards this goal, we proposed a new approach to accelerate this calculations for both simple and blind docking approaches by the exploitation of the emergent GPU architectures with the CUDA programming model and the hyper-Q technique. The latter enables multiple CPU threads to simultaneously launch multiple GPU Kernels to perform calculation.

We compared the performances of our parallel approach with the sequential version executed on a single-core CPU and with the parallel multi-core CPU version (48 CPU threads). Our parallel approach was applied to dock a set of protein-ligand complexes from the DUD database. The experiments show that the GPU-based parallel version outperforms both the multi-core CPU and sequential versions. Indeed, for 100 binding sites, our results show an average speedup of 186x compared to the serial implementation, and 10x as compared to the multi-core CPU version.

In addition, the experimental results show that the speedup of the GPU version increases exponentially with the number of binding sites.

In addition our parallel approach outperforms the best solution of the literature (MURCIA) since it runs faster than the CUDA version of that solution called (MURCIA-K1) and it is also faster than the second version called (MURCIA-K2) which uses CUDA with Trust and Particle systems libraries.

Scientific contributions

- Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernon, Horacio Emilio Pérez Sánchez, José M. Cecilia.(2020). Efficient GPU-based parallelization of solvation calculationfor the blind docking problem. J.Supercomput.76(3):1980-1998
- Hocine Saadi, Nadia Nouali-Taboudjemat, Malika Mehdi, OusmerSabrine. (2019). A GPU-MapCG based parallelization of BSO metaheuristic for Molecular Docking problem. The International Conference on Parallel and Distributed Processing Techniques Applications PDPTA'19:205-210 (USA).
- Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernon, Horacio Pérez Sánchez, José M. Cecilia. (2017). Parallel Desolvation Energy Term Calculation for Blind Docking on GPU Architectures. 46th International Conference on Parallel Processing Workshops. ICPP Workshops:16-22 (UK).
- H.Saadi, Y.Djenouri, N.Nouali-Taboudjemat, A.Rahmoun, M.Mehdi, A.Bendjoudi. (2016). Bees Swarm Optimization for Molecular Docking). The 6th International Conference on Metaheuristics and Nature Inspired Computing META'16 (Maroc)
- H.saadi, Y.Djanouri, N.Nouali-Taboudjemat, A.Rahmoune, M.Mehdi, A. Bendjoudi.(2016). Bees Swarm Optimisation for Molecular Docking (Poster). IWBBIO International Work-Conference on Bioinformatics and Biomedical Engineering. (Spain).

General Conclusion

In this thesis, we treated metaheuristics on GPU and their application to solving the molecular docking (MD) problem. We particularly addressed some challenging issues related to the MD problem such as the exploration of the huge search space by the appropriate metaheuristic and the evaluation of the results of the search step by a rapid and efficient scoring function. Also, we addressed the challenges related to the implementation of metaheuristics on modern GPU architectures such as CPU/GPU communication, GPU threads divergence, device hierarchically memory management etc. We leverage the computing power offered by GPUs through the use of GPGPU paradigm (General-purpose computing on GPUs). GPU can be found on workstations, clusters, personnel computing, and even on a smartphone. It becomes affordable and efficient with lower cost compared to the supercomputer.

We proposed three main contributions in this thesis. In the first contribution, we proposed a new approach named BSO-DOCK to efficiently explore the search space; this approach is based on Bees Swarm Optimization metaheuristic (BSO). We gave details of the operators of BSO

metaheuristique for MD, such as the encoding solution, the fitness function, the determination of regions, and the neighborhood search. The results revealed that BSO-DOCK showed its efficiency compared to the state-of-the-art algorithms. However, the performance decreases when dealing with very large instances, which requires a huge power computing.

In the second contribution, we intend to parallelize BSO-DOCK on GPUs architecture to overcome the bottleneck faced in the first contribution. The proposed approach is based on MapReduce Model. We parallelized the calculation of the scoring function of BSO-DOCK on GPU architecture with MapCG Framework. The latter allows us to write portable applications which can be executed on both CPU and GPU without changing any line of code. The results show a total speed-up exceeding 10x for the evaluation stage with respect to one CPU version.

In the third contribution, we efficiently parallelized the calculation of the most challenging part of the scoring function. The latter is the most consuming time part of the metaheuristic used to solve both simple and blind docking problems. We first developed a pure GPU oriented method using the CUDA programming model, we tuned our code to leverage the emergent GPUs architectures (Kepler, Maxwell, and Pascal). We then extend it to a hybrid method that simultaneously exploits both CPU and GPU cores by using the hyper-Q technique. The latter enables multiple CPU threads to simultaneously launch multiple GPU Kernel to perform the calculation. The different experiments demonstrate that the GPU-based parallel version outperforms both the multi-core CPU and the sequential versions. Indeed, for 100 binding sites, our results show an average speedup of 186x compared to the serial implementation, and 10x as compared to the multi-core CPU version. Our parallel approach outperforms the best solution of the literature.

Several conclusion can be drawn from our experience with the parallel metaheuristics on GPUs :

- The trend for parallel metaheuristic and parallel computing in the coming years is to use the GPU in cooperation with Many-core and Multi-core resources in what is known as heterogeneous parallel systems. The goal is to offer a huge computing resources for bioinformatics or other scientific fields.

- Developments of parallel metaheuristics for docking or other problems always focus on a specific platform. Thus, it is important to design parallel metaheuristics compatible with different hardware platforms.

Perspectives

As perspective to this work, we plane to :

- Design and implement the first Algerian platform for bioinformatics tools (molecular docking, molecular dynamics,etc.), this platform will leverage the high-performance computing resources offered by CERIST (Ibenbadis Cluster, GPUs stations etc). It will be available for all Algerian researchers to do rapid simulations for academics purpose for free.

- Apply the proposed parallel approaches in the context of virtual screening (a type of docking in which a database of ligands is docked with a target protein).
- Implement the proposed parallel algorithms on other architectures such as clusters with Multi-core and/or Mutli-GPU, explore the potential of vector unit architectures such as Xeon-Phy.
- Enhancing the scoring function quality by combining the solvation energy calculation presented in this thesis with other strategies like SASA (solvent-accessible surface area).
- Combine BSO with a local search metaheuristic such as Tabu search to batter explore the MD search space.
-

Bibliography

[Abbass 2001] Hussein A Abbass. *MBO: Marriage in honey bees optimization-A haplometrosis polygynous swarming approach*. In Proceedings of the 2001 Congress on evolutionary computation (IEEE Cat. No. 01TH8546), volume 1, pages 207–214. IEEE, 2001.

[Acar & Blelloch 2019] Umut A. Acar et Guy E. Blelloch. Algorithm design: Parallel and sequential. 2019.

[AUTOMOTIVE 2020] NVIDIA AUTOMOTIVE. *Giving Cars the Power to See, Think, and Learn*. http://www.nvidia.com/object/drive-automotive-technology.html, 2020.

[Ballester & Mitchell 2010] Pedro J Ballester et John BO Mitchell. *A machine learning approach to predicting protein–ligand binding affinity with applications to molecular docking*. Bioinformatics, vol. 26, no. 9, pages 1169–1175, 2010.

[Bleiweiss 2008] Avi Bleiweiss. *GPU accelerated pathfinding*. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pages 65–74. Eurographics Association, 2008.

[Board 2018] OpenMP Architecture Review Board. *The OpenMP Specification*. http://www.openmp.org, 2018. (accessed, April, 2th, 2017).

[Bozorg-Haddad *et al.* 2017] Omid Bozorg-Haddad, Mohammad Solgi et Hugo A Loáiciga. Meta-heuristic and evolutionary algorithms for engineering optimization. John Wiley & Sons, 2017.

[Bradley 2012] Thomas Bradley. *Hyper-Q example*. NVidia Corporation. Whitepaper v1. 0, 2012.

[Calégari 1999] Patrice Roger Calégari. *Parallelization of population-based evolutionary algorithms for combinatorial optimization problems*. Rapport technique, EPFL, 1999.

[Camacho & Vajda 2002] Carlos J Camacho et Sandor Vajda. *Protein–protein association kinetics and protein docking*. Current opinion in structural biology, vol. 12, no. 1, pages 36–40, 2002.

[Cecilia *et al.* 2013] José M Cecilia, José M García, Andy Nisbet, Martyn Amos et Manuel Ujaldón. *Enhancing data parallelism for ant colony optimization on GPUs*. Journal of Parallel and Distributed Computing, vol. 73, no. 1, pages 42–51, 2013.

[Chandrasekaran & Juckeland 2017] Sunita Chandrasekaran et Guido Juckeland. Openacc for programmers: Concepts and strategies. Addison-Wesley Professional, 2017.

[Chaudhary & Mishra 2016] Kamal Kumar Chaudhary et Nidhi Mishra. *A review on molecular docking: novel tool for drug discovery*. databases, vol. 3, no. 4, 2016.

[Codenotti & Leoncini 1992] Bruno Codenotti et Mauro Leoncini. Introduction to parallel processing. Addison-Wesley Longman Publishing Co., 1992.

[Cormen *et al.* 1994] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, Philippe Chrétienne et Xavier Cazin. Introduction à l'algorithmique. Dunod, 1994.

[Corporation 2018] NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE*. https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf, 2018.

[Dar & Mir 2017] Ayaz Mahmood Dar et S Mir. *Molecular docking: approaches, types, applications and basic challenges*. J Anal Bioanal Tech, vol. 8, no. 2, pages 1–3, 2017.

[DataBank 2017] RCSB DataBank. *Protein Data Bank*. https://http://www.rcsb.org/, 2017.

[Dean & Ghemawat 2008] Jeffrey Dean et Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. Communications of the ACM, vol. 51, no. 1, pages 107–113, 2008.

[Dean & Ghemawat 2010] Jeffrey Dean et Sanjay Ghemawat. *MapReduce: a flexible data processing tool*. Communications of the ACM, vol. 53, no. 1, pages 72–77, 2010.

[Delévacq *et al.* 2013] Audrey Delévacq, Pierre Delisle, Marc Gravel et Michaël Krajecki. *Parallel ant colony optimization on graphics processing units*. Journal of Parallel and Distributed Computing, vol. 73, no. 1, pages 52–61, 2013.

[Djenouri *et al.* 2018] Youcef Djenouri, Asma Belhadi et Riadh Belkebir. *Bees swarm optimization guided by data mining techniques for document information retrieval*. Expert Systems with Applications, vol. 94, pages 126–136, 2018.

[Dong *et al.* 2018] Dong Dong, Zhijian Xu, Wu Zhong et Shaoliang Peng. *Parallelization of molecular docking: A review*. Current Topics in Medicinal Chemistry, vol. 18, no. 12, pages 1015–1028, 2018.

[Drias *et al.* 2005] Habiba Drias, Souhila Sadeg et Safa Yahi. *Cooperative bees swarm for solving the maximum weighted satisfiability problem*. In International Work-Conference on Artificial Neural Networks, pages 318–325. Springer, 2005.

[Eisenberg & McLachlan 1986] David Eisenberg et Andrew D McLachlan. *Solvation energy in protein folding and binding*. Nature, vol. 319, no. 6050, pages 199–203, 1986.

[El-Hachem *et al.* 2017] Nehme El-Hachem, Benjamin Haibe-Kains, Athar Khalil, Firas H Kobeissy et Georges Nemer. *AutoDock and AutoDockTools for protein-ligand docking: Beta-site amyloid precursor protein cleaving enzyme 1 (BACE1) as a case study*. In Neuroproteomics, pages 391–403. Springer, 2017.

[Essaid *et al.* 2019] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot et Mathieu Brévilliers. *GPU parallelization strategies for metaheuristics: a survey*. International Journal of Parallel, Emergent and Distributed Systems, vol. 34, no. 5, pages 497–522, 2019.

[Ewing *et al.* 2001] Todd JA Ewing, Shingo Makino, A Geoffrey Skillman et Irwin D Kuntz. *DOCK 4.0: search strategies for automated molecular docking of flexible molecule databases*. Journal of computer-aided molecular design, vol. 15, no. 5, pages 411–428, 2001.

[Fan *et al.* 2019] Jiyu Fan, Ailing Fu et Le Zhang. *Progress in molecular docking*. Quantitative Biology, pages 1–7, 2019.

[Fang *et al.* 2014] Jianbin Fang, Ana Lucia Varbanescu, Baldomero Imbernon, José M Cecilia et Horacio Emilio Pérez Sánchez. *Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi.* In IWBBIO, pages 579–588, 2014.

[Fang *et al.* 2016] Ye Fang, Yun Ding, Wei P Feinstein, David M Koppelman, Juana Moreno, Mark Jarrell, J Ramanujam et Michal Brylinski. *GeauxDock: Accelerating structure-based virtual screening with heterogeneous computing*. PloS one, vol. 11, no. 7, page e0158898, 2016.

[Farber 2016] Rob Farber. Parallel programming with openacc. Newnes, 2016.

[Flynn & Rudd 1996] Michael J Flynn et Kevin W Rudd. *Parallel architectures*. ACM Computing Surveys (CSUR), vol. 28, no. 1, pages 67–70, 1996.

[Flynn 1966] Michael J Flynn. *Very high-speed computing systems*. Proceedings of the IEEE, vol. 54, no. 12, pages 1901–1909, 1966.

[Flynn 1995] Michael J Flynn. Computer architecture: Pipelined and parallel processor design. Jones & Bartlett Learning, 1995.

[Forli *et al.* 2012] W Forli, Scott Halliday, Rik Belew et Arthur J Olson. *AutoDock Version 4.2*, 2012.

[Geshi 2019] Masaaki Geshi. The art of high performance computing for computational science, vol. 2. Springer, 2019.

[Ghorpade *et al.* 2012] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni et Amit Bawaskar. *GPGPU processing in CUDA architecture*. arXiv preprint arXiv:1202.4347, 2012.

[GPGPU 2017] GPGPU. *CUDA white paper zone*. https://developer.nvidia.com/cuda-zone, 2017.

[Green 2010] Simon Green. *Particle simulation using cuda*. NVIDIA whitepaper, vol. 6, pages 121–128, 2010.

[Guerrero *et al.* 2011] Ginés D Guerrero, Horacio Pérez-Sánchez, Wolfgang Wenzel, José M Cecilia et José M Garcia. *Effective parallelization of non-bonded interactions kernel for virtual screening on gpus*. In 5th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB 2011), pages 63–69. Springer, 2011.

[Guerrero *et al.* 2012] Gines D Guerrero, Horacio E Perez-S, Jose M Cecilia, Jose M Garcia*et al. Parallelization of virtual screening in drug discovery on massively parallel architectures*. In 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pages 588–595. IEEE, 2012.

[Guo *et al.* 2014] Liyong Guo, Zhiqiang Yan, Xiliang Zheng, Liang Hu, Yongliang Yang et Jin Wang. *A comparison of various optimization algorithms of protein–ligand docking programs by fitness accuracy*. Journal of molecular modeling, vol. 20, no. 7, page 2251, 2014.

[Halperin *et al.* 2002] Inbal Halperin, Buyong Ma, Haim Wolfson et Ruth Nussinov. *Principles of docking: An overview of search algorithms and a guide to scoring functions*. Proteins: Structure, Function, and Bioinformatics, vol. 47, no. 4, pages 409–443, 2002.

[Hetenyi & van der Spoel 2006] Csaba Hetenyi et David van der Spoel. *Blind docking of drug-sized compounds to proteins with up to a thousand residues*. FEBS Letters, vol. 580, no. 5, pages 1447–1450, 2006.

[Hetényi & van der Spoel 2002] Csaba Hetényi et David van der Spoel. *Efficient docking of peptides to proteins without prior knowledge of the binding site*. Protein Science, vol. 11, no. 7, pages 1729–1737, 2002.

[Hong *et al.* 2010] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng et Haibo Lin. *MapCG: writing parallel program portable between CPU and GPU*. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pages 217–226. ACM, 2010.

[Hong *et al.* 2012] Chun-Tao Hong, De-Hao Chen, Yu-Bei Chen, Wen-Guang Chen, Wei-Min Zheng et Hai-Bo Lin. *Providing source code level portability between CPU and GPU with MapCG*. Journal of Computer Science and Technology, vol. 27, no. 1, pages 42–56, 2012.

[Huang & Zou 2010] Sheng-You Huang et Xiaoqin Zou. *Advances and challenges in protein-ligand docking*. International journal of molecular sciences, vol. 11, no. 8, pages 3016–3034, 2010.

[Huang *et al.* 2006] Niu Huang, Brian K. Shoichet et John J. Irwin. *Benchmarking Sets for Molecular Docking*. Journal of Medicinal Chemistry, vol. 49, no. 23, pages 6789–6801, 2006. PMID: 17154509.

[Huey *et al.* 2007] Ruth Huey, Garrett M Morris, Arthur J Olson et David S Goodsell. *A semiempirical free energy force field with charge-based desolvation*. Journal of computational chemistry, vol. 28, no. 6, pages 1145–1152, 2007.

[Imbernón *et al.* 2018] Baldomero Imbernón, Javier Prades, Domingo Giménez, José M Cecilia et Federico Silla. *Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rCUDA study*. Future Generation Computer Systems, vol. 79, pages 26–37, 2018.

[Irwin & Shoichet 2017] Irwin et Shoichet. *Ligand Data Bank*. http://zinc.docking.org/, 2017.

[JéJé 1992] Joseph JéJé. An introduction to parallel algorithms. Reading, MA: Addison-Wesley, 1992.

[Kannan & Ganji 2010] S. Kannan et R. Ganji. *Porting Autodock to CUDA*. In IEEE Congress on Evolutionary Computation, pages 1–8, July 2010.

[Karaboga & Akay 2009] Dervis Karaboga et Bahriye Akay. *A survey: algorithms simulating bee swarm intelligence*. Artificial intelligence review, vol. 31, no. 1-4, page 61, 2009.

[Karaboga 2005] Dervis Karaboga. *An idea based on honey bee swarm for numerical optimization*. Rapport technique, Technical report-tro6, Erciyes university, engineering faculty, computer . . . , 2005.

[Kinnings *et al.* 2011] Sarah L Kinnings, Nina Liu, Peter J Tonge, Richard M Jackson, Lei Xie et Philip E Bourne. *A machine learning-based method to improve docking scoring functions and its application to drug repurposing*. Journal of chemical information and modeling, vol. 51, no. 2, pages 408–419, 2011.

[Kirk & Wen-Mei 2016] David B Kirk et W Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2016.

[Lang *et al.* 2009] P Therese Lang, Scott R Brozell, Sudipto Mukherjee, Eric F Pettersen, Elaine C Meng, Veena Thomas, Robert C Rizzo, David A Case, Thomas L James et Irwin D Kuntz. *DOCK 6: Combining techniques to model RNA–small molecule complexes*. Rna, vol. 15, no. 6, pages 1219–1230, 2009.

[Liu & Wang 1999] Ming Liu et Shaomeng Wang. *MCDOCK: a Monte Carlo simulation approach to the molecular docking problem*. Journal of computer-aided molecular design, vol. 13, no. 5, pages 435–451, 1999.

[Liu & Wang 2015] Jie Liu et Renxiao Wang. *Classification of current scoring functions*. Journal of chemical information and modeling, vol. 55, no. 3, pages 475–482, 2015.

[Liu *et al.* 2005] Bo-Fu Liu, Hung-Ming Chen, Hui-Ling Huang, Shiow-Fen Hwang et Shinn-Ying Ho. *Flexible protein-ligand docking using particle swarm optimization*. In 2005 IEEE Congress on Evolutionary Computation, volume 1, pages 251–258. IEEE, 2005.

[Liu *et al.* 2009] Yu Liu, Wentao Li, Yongliang Wang et Mingwei Lv. *An efficient approach for flexible docking base on particle swarm optimization*. In 2009 2nd International Conference on Biomedical Engineering and Informatics, pages 1–7. IEEE, 2009.

[Liu *et al.* 2017] Lifeng Liu, Yue Zhang, Meilin Liu, Chongjun Wang et Jun Wang. *A-MapCG: An Adaptive MapReduce Framework for GPUs*. In Networking, Architecture, and Storage (NAS), 2017 International Conference on, pages 1–8. IEEE, 2017.

[Lopez-Camacho *et al.*] Esteban Lopez-Camacho, Maria Jesus Garcia Godoy, Jose Garcia-Nieto, Antonio J Nebro et Jose Aldana-Montes. *Solving molecular flexible docking problems with metaheuristics: A comparative study*.

[López-Camacho *et al.* 2014] Esteban López-Camacho, María Jesús García Godoy, Antonio J Nebro et José F Aldana-Montes. *jMetalCpp: optimizing molecular docking problems with a C++ metaheuristic framework*. Bioinformatics, vol. 30, no. 3, pages 437–438, 2014.

[Matloff 2006] Norman Matloff. *Introduction to parallel processing*, 2006.

[Mehdi *et al.* 2013] Malika Mehdi, Ahcene Bendjoudi, Lakhdar Loukil et Nouredine Melab. *Parallel GPU-accelerated metaheuristics*. Designing scientific applications on GPUs, vol. 183, 2013.

[Morris & Lim-Wilby 2008] Garrett M Morris et Marguerita Lim-Wilby. *Molecular docking*. In Molecular modeling of proteins, pages 365–382. Springer, 2008.

[Morris *et al.* 1998a] Garrett M Morris, David S Goodsell, Robert S Halliday, Ruth Huey, William E Hart, Richard K Belew et Arthur J Olson. *Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function*. Journal of computational chemistry, vol. 19, no. 14, pages 1639–1662, 1998.

[Morris *et al.* 1998b] Garrett M Morris, David S Goodsell, Robert S Halliday, Ruth Huey, William E Hart, Richard K Belew, Arthur J Olson*et al. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function*. Journal of computational chemistry, vol. 19, no. 14, pages 1639–1662, 1998.

[Morris *et al.* 2009] Garrett M Morris, Ruth Huey, William Lindstrom, Michel F Sanner, Richard K Belew, David S Goodsell et Arthur J Olson. *AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility*. Journal of computational chemistry, vol. 30, no. 16, pages 2785–2791, 2009.

[Morris *et al.* 2010] Garrett M Morris, David S Goodsell, ME Pique, WL Lindstrom, R Huey, S Forli, WE Hart, S Halliday, R Belew et AJ Olson. *Automated Docking of Flexible Ligands to Flexible Receptors*. user guide AutoDock, 2010.

[Mukesh & Rakesh 2011] Backwani Mukesh et Kumar Rakesh. *Molecular docking: a review*. Int J Res Ayurveda Pharm, vol. 2, pages 746–1751, 2011.

[Namasivayam & Günther 2007] Vigneshwaran Namasivayam et Robert Günther. *PSO@ AUTODOCK: A fast flexible molecular docking program based on swarm intelligence*. Chemical biology & drug design, vol. 70, no. 6, pages 475–484, 2007.

[National Institutes of Health (NIH) 017] National Institutes of Health (NIH). *Ligand Data Bank*. https://pubchem.ncbi.nlm.nih.gov/compound/Ligand/, 2017.

[Nickolls & Dally 2010] J. Nickolls et W. J. Dally. *The GPU Computing Era*. IEEE Micro, vol. 30, no. 2, pages 56–69, March 2010.

[NVIDIA CUDA 2017] NVIDIA CUDA. *CUDA C BEST PRACTICES GUIDE*. https://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf, 2017.

[NVIDIA 2017] NVIDIA. *Thrust*. http://docs.nvidia.com/cuda/thrust/index.html, 2017.

[NVIDIA 2018] NVIDIA. *Pascal Architecture*. https://devblogs.nvidia.com/inside-pascal/, 2018.

[Owens *et al.* 2008] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone et James C Phillips. *GPU computing*. Proceedings of the IEEE, vol. 96, no. 5, pages 879–899, 2008.

[Pagadala *et al.* 2017] Nataraj S Pagadala, Khajamohiddin Syed et Jack Tuszynski. *Software for molecular docking: a review*. Biophysical reviews, vol. 9, no. 2, pages 91–102, 2017.

[Pechan & Feher 2011] Imre Pechan et Bela Feher. *Molecular docking on FPGA and GPU platforms*. In Field Programmable Logic and Applications (FPL), 2011 International Conference on, pages 474–477. IEEE, 2011.

[Pettersen *et al.* 2004] Eric F Pettersen, Thomas D Goddard, Conrad C Huang, Gregory S Couch, Daniel M Greenblatt, Elaine C Meng et Thomas E Ferrin. *UCSF Chimera—a visualization system for exploratory research and analysis*. Journal of computational chemistry, vol. 25, no. 13, pages 1605–1612, 2004.

[Pham *et al.* 2005] DT Pham, A Ghanbarzadeh, E Koc, S Otri, S Rahim et M Zaidi. *The bees algorithm*. Technical Note, Manufacturing Engineering Centre, Cardiff University, UK, 2005.

[Raschka 2014] S Raschka. *Molecular docking, estimating free energies of binding, and AutoDock's semi-empirical force field*, 2014.

[Rawal *et al.* 2019] Kamal Rawal, Tanishka Khurana, Himanshu Sharma, Sadika Verma, Simmi Gupta, Chahat Kubba, Ulrich Strych, Peter Jay Hotez et Maria Elena Bottazzi. *An extensive survey of molecular docking tools and their applications using text mining and deep curation strategies.* PeerJ Preprints, vol. 7, page e27538v1, 2019.

[Renli *et al.* 2016] Wang Renli, Dai Yueming et Dong Liming. *The application of Apriori-BSO algorithms in medical records data mining*. In 2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference, pages 827–832. IEEE, 2016.

[Saadi *et al.* 2016] H Saadi, Y Djenouri, N Nouali, A Rahmoun, M Mehdi et A Bendjoudi. *Bees Swarm Optimization for Molecular Docking*. In The 6th International Conference on Metaheuristics and Nature Inspired-Computing META'16, pages 84–89. sciencesconf.org, 2016.

[Saadi *et al.* 2017] Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernón, Horacio Peréz-Sánchez et José M Cecilia. *Parallel desolvation energy term calculation for blind docking on GPU architectures.* In Parallel Processing Workshops (ICPPW), 2017 46th International Conference on, pages 16–22. IEEE, 2017.

[Saadi *et al.* 2019a] Hocine Saadi, Nadia Nouali Taboudjemat et Malika Mehdi. *A GPU-MapCG based parallelization of BSO metaheuristic for Molecular Docking problem*. In PDPTA'19 - The 25th Int'l Conf on Parallel and Distributed Processing Techniques and Applications, pages 205–210, August 2019.

[Saadi *et al.* 2019b] Hocine Saadi, Nadia Nouali Taboudjemat, Abdellatif Rahmoun, Horacio Pérez-Sánchez, Jose M Ceciliaet *al. Efficient GPU-based parallelization of solvation calculation for the blind docking problem*. The Journal of Supercomputing, pages 1–19, 2019.

[Sadeg & Drias 2007] Souhila Sadeg et Habiba Drias. *A selective approach to parallelise Bees Swarm Optimisation metaheuristic: application to MAX-W-SAT*. International Journal of Innovative Computing and Applications, vol. 1, no. 2, pages 146–158, 2007.

[Sanders & Kandrot 2010] Jason Sanders et Edward Kandrot. Cuda by example: An introduction to general-purpose gpu programming, portable documents. Addison-Wesley Professional, 2010.

[Science & Imaging 2020] NVIDIA Science et Medical Imaging. *Accelerating Science and Medical Imaging with NVIDIA GPUS -/science-and-medical*. http://www.nvidia.com/object/science-and-medical-imaging.html, 2020.

[Scripps-Research-Inst 2009] Scripps-Research-Inst. *Autodock Equations*. http://autodock.scripps.edu/resources/science/equations/, 2009.

[Seeliger & De Groot 2010] Daniel Seeliger et Bert L De Groot. *Ligand docking and binding site analysis with PyMOL and Autodock/Vina*. Journal of computer-aided molecular design, vol. 24, no. 5, pages 417–422, 2010.

[Shi *et al.* 2018] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu et Qiang-Sheng Hua. *Graph processing on GPUs: A survey*. ACM Computing Surveys (CSUR), vol. 50, no. 6, pages 1–35, 2018.

[Silva *et al.* 2014] Cândida G Silva, Pedro Carreiras, Elsa Henriques, Carlos JV Simoes et Rui MM Brito. *A Machine Learning Approach to Enhance Scoring Performance in Docking-Based Virtual Screening Experiments: COX-1 as a Case Study.* In IWBBIO, pages 406–414, 2014.

[Simonsen *et al.* 2011] Martin Simonsen, Christian NS Pedersen, Mikael H Christensen et René Thomsen. *GPU-accelerated high-accuracy molecular docking using guided differential evolution: real world applications*. In Proceedings of the 13th annual conference on genetic and evolutionary computation, pages 1803–1810, 2011.

[Sousa S.F & M.J 2006] Fernandes P.A Sousa S.F et Ramos M.J. *Protein–ligand docking: Current status and future challenges*. Proteins: Structure, Function, and Bioinformatics, 2006.

[Sousa *et al.* 2006] Sergio Filipe Sousa, Pedro Alexandrino Fernandes et Maria Joao Ramos. *Protein–ligand docking: current status and future challenges*. Proteins: Structure, Function, and Bioinformatics, vol. 65, no. 1, pages 15–26, 2006.

[Sukhwani & Herbordt 2009] Bharat Sukhwani et Martin C Herbordt. *GPU acceleration of a production molecular docking code*. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pages 19–27, 2009.

[Talbi 2009] El-Ghazali Talbi. Metaheuristics: from design to implementation, volume 74. John Wiley & Sons, 2009.

[Teodorovic & Dell'Orco 2005] Dušan Teodorovic et Mauro Dell'Orco. *Bee colony optimization–a cooperative learning approach to complex transportation problems*. Advanced OR and AI methods in transportation, vol. 51, page 60, 2005.

[Trobec *et al.* 2018] Roman Trobec, Boštjan Slivnik, Patricio Bulić et Borut Robič. Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms. Springer, 2018.

[Trott & Olson 2010] Oleg Trott et Arthur J Olson. *AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading*. Journal of computational chemistry, vol. 31, no. 2, pages 455–461, 2010.

[TSRI 2017] The Scripps Research Institute. TSRI. *Desolvation Free Energy Term in AutoDock 4*. http:

//autodock.scripps.edu/resources/science/autodock-4-desolvation-free-energy//, 2017.

[Verdonk *et al.* 2003] Marcel L Verdonk, Jason C Cole, Michael J Hartshorn, Christopher W Murray et Richard D Taylor. *Improved protein ligand docking using GOLD*. Proteins: Structure, Function, and Bioinformatics, vol. 52, no. 4, pages 609–623, 2003.

[Vitali *et al.* 2019] Emanuele Vitali, Davide Gadioli, Andrea Beccari, Carlo Cavazzoni, Cristina Silvano et Gianluca Palermo. *An hybrid approach to accelerate a molecular docking application for virtual screening in heterogeneous nodes: POSTER*. In Proceedings of the 16th ACM International Conference on Computing Frontiers, pages 298–299, 2019.

[Wedde *et al.* 2004] Horst F Wedde, Muddassar Farooq et Yue Zhang. *Beehive: An efficient fault-tolerant routing algorithm inspired by honey bee behavior*. In International Workshop on Ant Colony Optimization and Swarm Intelligence, pages 83–94. Springer, 2004.

[Whitepaper] NVIDIA Whitepaper. *NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built*.

[Whitepaper 2018] NVIDIA Whitepaper. *NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built*, 2018.

[wiki Bees 2020] wiki Bees. *Honey Bee*. https://en.wikipedia.org/wiki/Honey_bee, 2020.

[Yan *et al.* 2017] Yumeng Yan, Di Zhang, Pei Zhou, Botong Li et Sheng-You Huang. *HDOCK: a web server for protein–protein and protein–DNA/RNA docking based on a hybrid strategy*. Nucleic acids research, vol. 45, no. W1, pages W365–W373, 2017.

[Zhang *et al.* 2013] Qian Zhang, Junmei Wang, Gines D Guerrero, Jose M Cecilia, Jose M García, Youyong Li, Horacio Perez-Sanchez et Tingjun Hou. *Accelerated conformational entropy calculations using graphic processing units*. Journal of chemical information and modeling, vol. 53, no. 8, pages 2057–2064, 2013.

الملخص

الالتحام الجزيئي هي تقنية تستخدم في صناعة الأدوية في عملية اكتشاف عقاقير جديدة. الهدف هو التنبؤ بوضية الارتباطبين جزيىء صغير (مرشح دواء) وهدف بروتيني (أصل المرض). نظرًا لأن هذه المشكلة معقدة ومعروفة بأنها صعبة فقد تم اقتراح العديد من الخوارزميات الوصفية لحلها ، مثل الخوارزمية الجينية، وتحسين سرب الجسيمات، وما إلى ذلك ، ومع ذلك، فإن هاته الخورزميات تتطلب قدرًا كبيرًا من موارد الحوسبة. في الوقت الحاضر، يمكن توفير قوة الحساب هاته بواسطة وحدة معالجة الرسومات من خلال استخدام نموذج الحوسبة للأغراض العامة على وحدات معالجة الرسومات. تقدم هذه الأطروحة مقاربتين لموزاة الخورزميات الوصفية على وحدة معالج الرسومات لحل مشكلة الالتحام الجزيئي ومن ثم تقييمها من حيث وقت الحساب الذي تم تحقيقه. أثبتت المقاربات المقترحة فعاليتها في تسريع لالتحام الجزيئي على وحدات معالجة الرسومات عند مقارتها بوحدة المعالجة المركزية أحادية النواة أو وحدة المعالجة المركزية متعددة النواة. إلى جانب تقديم مقاربات الموازاة، نقترح أيضًا خورازمية وصفية جديدة تعتمد على خوارزمية سرب النحل لحل مشكلة الالتحام الجزيئي كبديل للخوارزميات الوصفية التقليدية مثل الخوارزمية الجينية.

Résumé

L'Amarrage Moléculaire (AM) est une technique utilisée dans l'industrie pharmaceutique dans le process de découverte de nouveaux médicaments. L'objectif est de prédire la position d'interaction entre une petite molécule (substance médicale) et une protéine cible (l'origine d'une maladie). Ce problème est complexe et connu pour être NP-difficile. Plusieurs métaheuristiques ont été proposées pour le résoudre, telles que l'algorithme génétique (GA), l'optimisation d'essaims de particules (PSO), etc. Néanmoins, ces métaheuristiques sont gourmandes en temps de calcul et souvent nécessitent beaucoup de ressources de calcul informatiques. Aujourd'hui, cette puissance peut être fournie par les carte de traitement graphique (GPU) et grâce à l'utilisation du paradigme GPGPU (General Purpose Computing on GPU). Cette thèse présente deux approches de parallélisation des métaheuristiques sur les cartes GPU pour résoudre le problème d'amarrage moléculaire, ainssi une évaluation adéquate en termes de temps de calcul réalisé sur différentes architectures GPUs, Les approches de parallélisation proposée ont prouvé leur efficacité dans l'accélération de AM sur les GPUs, ces résultats on été comparés avec un seul et multi-coeur CPU. En plus de la présentation des stratégies de parallélisation, nous proposons également un nouvau algorithme basé sur la metaheuristique d'essaim d'abeilles (BSO) comme alternative aux métaheurismes traditionnels tels que les algorithmes GA et PSO pour résoudre le problème d' Amarrage Moléculaire.

Abstract

Molecular Docking (MD) is a technique used in the pharmaceutical industry in the process of discovering new drugs. The aim is to predict the binding pose between a small molecule (drug candidate) and a protein target (the origin of a disease). Since this problem is complex and known to be NP-hard, several metaheuristics have been proposed to solve it, such as Genetic Algorithm (GA), and Particle Swarm Optimization (PSO), ... etc. Nevertheless, these metaheuristics are greedy in terms of computation time and often require a huge amount of computing resources. Nowadays, such computation power can be provided by the Graphics Processing Unit (GPUs) through the use of the GPGPU paradigm (General-purpose computing on GPUs). This thesis presents two parallelization approaches of a metaheuristic on GPU to solve the molecular docking problem, and a corresponding evaluation in terms of computation time achieved on different NVIDIA GPU architectures. The parallel metaheuristic approaches have proven their effectiveness in accelerating MD on GPUs when compared to a single-core CPU or multi-core CPU. Besides presenting the parallelization strategies, we also propose a new metaheuristic based on bees swarm optimization (BSO) algorithm to solve the MD problem as an alternative to the traditional metaheuristics such as GA and PSO algorithms.

Keywords: Metaheuristics, GPUs, BSO, Molecular Docking.