POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

High Throughput Molecular Expansion for Extreme Scale Virtual Screening Campaign

Tesi di Laurea Magistrale in
High Performance Computing Engineering - Ingegneria del Calcolo ad Alte Prestazioni

Author: **Giuseppe Gentile**

Student ID: 222268
Advisor: Prof. Gianluca Palermo
Co-advisors: Davide Gadioli, Gianmarco Accordi
Academic Year: 2023-24

Abstract

In drug discovery, the search for treatments traditionally involves in vitro testing of chemical compounds against target proteins, but this approach is often slow, costly, and resource-intensive. The advent of supercomputing has revolutionized this process by allowing computational models to virtually predict promising compounds for experimental testing. This has led to the creation of virtual screening pipelines, where large datasets of chemical compounds are computationally examined to identify potential drug candidates. Chemical libraries have expanded to millions of compounds, requiring High-Performance Computing (HPC) infrastructures to efficiently process such volumes. A key challenge is balancing computational performance with storage demands. Precomputing the minimum-energy conformations of molecules can speed up screening, but storing the 3D structures of millions of molecules requires enormous storage capacity, often reaching petabytes, which is impractical. Achieving a balance between performance and storage is crucial. Compact molecular representations, such as the Simplified Molecular Input Line Entry System (SMILES), reduce storage requirements compared to 3D structures. From SMILES, a low-energy stable 3D conformation can be generated, which is essential for accurate drug screening. Energy minimization is the central computational challenge. This thesis explores optimizations to minimize the energy of molecules derived from their SMILES representation. By integrating techniques like parallelization to efficiently expand compact molecular storage, we aim to accelerate pre-processing for virtual screening, enabling faster and scalable exploration of large chemical datasets and increasing the likelihood of discovering new drugs.

Keywords: High Performance Computing, Ligand, GPU, parallelization

Abstract in lingua italiana

Nella scoperta di farmaci, la ricerca di trattamenti tradizionalmente coinvolge test in vitro di composti chimici contro proteine bersaglio, ma questo approccio è spesso lento, costoso e richiede molte risorse. L'avvento del supercalcolo ha rivoluzionato il processo, consentendo ai modelli computazionali di prevedere virtualmente composti promettenti da testare sperimentalmente. Questo ha portato alla creazione di pipeline di screening virtuale, dove vasti dataset di composti chimici vengono esaminati per identificare potenziali farmaci. Le librerie chimiche sono cresciute fino a milioni di composti, richiedendo infrastrutture HPC (High-Performance Computing) per elaborare efficientemente tali volumi. Una sfida chiave è bilanciare prestazioni computazionali e archiviazione. Precalcolare le conformazioni a energia minima delle molecole può accelerare lo screening, ma memorizzare le strutture 3D di milioni di molecole richiede enormi capacità di archiviazione, fino a petabyte, spesso impraticabili. Un equilibrio tra prestazioni e archiviazione è cruciale. Rappresentazioni molecolari compatte, come il Simplified Molecular Input Line Entry System (SMILES), riducono i requisiti di archiviazione rispetto alle strutture 3D. Dagli SMILES si può generare una conformazione 3D stabile a bassa energia, fondamentale per lo screening farmacologico. La minimizzazione dell'energia è la sfida computazionale principale. Questa tesi esplora l'uso di ottimizzazioni per minimizzare l'energia delle molecole derivate dalla loro rappresentazione SMILES. Integrando tecniche come la parallelizzazione con una archiviazione molecolare compatta, miriamo ad accelerare la fase di preprocessing dello screening virtuale, consentendo un'esplorazione più rapida e scalabile di grandi dataset chimici, aumentando la probabilità di scoprire nuovi farmaci.

Parole chiave: Calcolo ad Alte Prestazioni, Ligandi, GPU, parallelizzazione

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Background & State of the Art **5**

2.1 How to represent a molecule 5
2.2 Force Field 8
2.2.1 Empirical Force Fields 10
2.2.2 Merck Molecular Force Field 10
2.2.3 Machine Learning-based 18
2.3 Energy minimization 18
2.3.1 BFGS algorithm 20
2.3.2 L-BFGS algorithm 22
2.3.3 Energy Minimization Software 22
2.4 Virtual Screening 23
2.4.1 LiGen 23
2.5 HPC 25

3 Proposed Method **31**

3.1 CPU Optimization 31
3.1.1 Latency parallelization 31
3.1.2 Throughput parallelization 32
3.2 GPU parallelization 35
3.3 Minimizing work imbalance of bucket 42
3.4 Design kernel structure 45

3.5 Kernel Optimization 47
3.5.1 Non-bonded interaction problem 49
3.5.2 Optimizing non-bonded interactions 51
3.6 Parameter selection and final computational considerations 59

4 Results **61**
4.1 Experimental setup 61
4.1.1 Hardware description 61
4.1.2 Dataset 62
4.1.3 Profiled measures 62
4.2 CPU parallelization results 63
4.2.1 CPU Pipeline parallelization 63
4.3 GPU porting Analysis 66
4.3.1 Mixed precision Effect 66
4.3.2 Result of optimizing non bonded interactions 68
4.3.3 Bucketizer Strategy Analysis 70
4.4 Speedup CPU only vs CUDA integration 72

5 Conclusions **77**

Bibliography **79**

List of Figures **83**

List of Tables **85**

Acknowledgements **87**

1 | Introduction

The need to discover treatments has driven innovation in medicine over the centuries. In this field, milestones such as discovery of antibiotics, vaccines, and targeted therapies have been reached. Advances in chemistry, biology, and pharmacology have refined the methodologies used to discover new treatments, leading to the structured and systematic drug discovery process well organized in steps. It begins with target identification, where a biological module (such as a protein) plays a crucial role in a disease. Following this, the lead discovery phase seeks chemical compounds that interact with the previously identified target in such a way that the disease's effect are suppressed. Prior to last decade, this was achieved by testing several compounds (either existing ones or engineered using chemistry knowledge) against the biological target, measuring their effectiveness. Then promising candidates are refined through lead optimization, a steps that try to enhance efficacy and reduce their potential toxicity. Once the best compound of the tested set is identified, it undergoes preclinical testing, where its safety and effectiveness are evaluated in vitro (cell-based assays) and in vivo (animal models). Finally, after passing preclinical evaluation, the drug enters clinical trials, involving humans to ensure efficacy and safety before regulatory approval and commercial distribution. While target identification, lead optimization and preclinical testing stages requires high level of medical and biology knowledge, as well as adherence to ethical guidelines, lead discovery relies on a trial and error approach that can be automatized and simulated. Moreover, experimenting on laboratories is resource intensive, requiring materials and hours of human work. Traditional approaches require vast compound libraries and experimental effort to identify drug candidates. The costs associated with synthesizing and testing each compound *in vitro* are substantial, making the discovery process expensive and inefficient. Furthermore, despite significant advancements, the majority of initially promising compounds fails at later stages due to unexpected toxicity, poor bioavailability, or off-target effects [19]. Researchers put efforts in trying to simulate the lead discovery phase rather that executing it in laboratory, aiming to performing *in vivo* techniques in combinations to prior exploration phase with *in silico* simulations, that would lead to lower cost of productions. Already in past century, computational methods that aim to simulate biochemical systems emerged, fully estab-

lishing their foundation in 1981 [26]. The simulation of the interactions between molecule and target algorithmically was formulated in the 80s as well [12], and opened a research world called docking. However, at that time, computing power was limited and simulate big compound libraries against a target was infeasible. Recently, advancements in high-performance computing (HPC) enabled large-scale molecular simulations. Modern HPC architectures allow researchers to analyze billions of molecules in a matter of days, increasing the efficiency of early-stage drug discovery. For instance, the EXSCALATE platform demonstrated the ability to screen 70 billion molecules against SARS-CoV-2 targets in just 60 hours [9]. Despite these advances, new computational bottlenecks continue to emerge, particularly concerning the preparation and handling of molecular data at an extreme scale. A major challenge in virtual screening is the need to handle storage of 3D molecular conformations for docking simulations. Docking simulations require 3D coordinates of molecule to model ligand-target interactions. The conventional approach involves precomputing and storing these conformers for large compound libraries, but as the scale of those libraries increases to billions of compounds (as in the EXSCALATE experiment), this becomes infeasible due to prohibitive storage requirements. Alternatively, generating conformers dynamically during virtual screening introduces latencies that reduce the performance gains achieved by HPC docking, since generating molecular stable conformations algorithmically is itself a complex task too. Existing tools such as **RDKit** [21] provide solutions for very accurate conformation generation, yet they face scalability limitations when dealing with large molecular libraries. Efforts to improve performance have largely focused on optimizing conformation generation for single molecules or small batches, as seen in frameworks like OpenMM [7]. However, when applied at extreme scales, these methods become computational bottlenecks rather than accelerators. Furthermore, approximation techniques aimed at speeding up conformer generation often introduce distortions in molecular geometry, leading to docking inaccuracies and an increase in false positives [8]. As a result, there is the need for a method that enables fast, accurate, and scalable (to HPC infrastructures) generation of 3D molecular conformations within a virtual screening pipeline. For this reason, this thesis proposes a scalable framework for SMILES-to-conformation expansion, optimized for integration with an existing HPC-driven virtual screening pipeline: LiGen. The primary objective of the proposed method is to eliminate the storage bottleneck caused by saving precomputed stable conformers, while maintaining docking accuracy, preserving the quality of conformers generated in the original version. To reach the desired objective, the original application has first been analyzed, looking for parallelization opportunities. Being the expansion of molecules a modular algorithm, each task have been analyzed both in isolation to understand its complexness, and as a whole to understand the possible chain effect of

one stage's performance over the others. Moreover, the application is modular, with each modules performing complex operations quite different from other modules. At the same time, target architectures are as well modular and heterogeneous, capable of handling different computational workloads. Application and its target architectures shares this concept of heterogeneity, hence the mapping of algorithm and resources becomes almost immediate. More complex stages will have allocated more computational resources, while lighter ones are scaled down. Work of this thesis consisted also in handling and analyze the effect of this levels of different parallelization granularity. Also, different hardware used for parallelizing different problem requires different solutions and computational organization. Reorganization of the algorithm to be efficient on the responsible resources is where the major effort of the work of this thesis was spent. After studying and grasping available architectures working mechanism, optimizations in the direction of exploiting the hardware are implemented.

In order to grasp the work of this thesis, broad concepts are required to be understood. An introduction to general computational chemistry, optimization algorithms and architectural review is provided in Chapter 2, together with state-of-the-art solution to the expansion of molecule problem, illustrating main theoretical results obtained so far. The proposed methodology implemented to solve the problem is then introduced in Chapter 3, showing how the computation is organized, motivating any implementation choices. Chapter 4 reviews the obtained results and analysis of implemented optimizations. Finally, Chapter 5 concludes the work and illustrates future works with suggestion to further improve this thesis's results.

2 | Background & State of the Art

To fully comprehend the topics discussed in this thesis, is essential to grasp foundational concepts that span from basic chemistry, biology, and computational methodologies. This chapter is dedicated to introducing these key concepts, providing the necessary background knowledge that ensure a clear understanding of the work.

Additionally, this chapter includes an overview of existing projects with similar objectives, comparing their approaches and final goals to highlight both similarities and differences.

Existing Molecular Dynamics softwares [7, 20] comes also with energy minimizer modules incorporated. However, these tools primarily focus on optimizing the energy of individual molecular systems, leveraging GPU acceleration to enhance performance as the system size increases.

The focus of this thesis, in contrast, is to accelerate the energy minimization of a large number of molecules, rather than a single system. This presents a unique challenge that has not been addressed by existing software. Our approach investigates how High-Performance Computing (HPC) techniques can be applied to this problem, enabling parallelization to handle multiple molecules simultaneously. For this reason, a section explaining basic concept of parallelization and architectures is included in this chapter.

Moreover, in contrast to typical existing software, which performs energy minimization directly on molecular structures, the approach presented in this thesis encompasses the entire process, from the initial molecular structure generation from SMILES to energy minimization. To the best of our knowledge, there is currently no software that directly converts a SMILES string into a minimum energy molecular conformation.

2.1. How to represent a molecule

A **ligand** is defined as a small molecule that binds to a larger target molecule, typically a protein, to form a stable complex. Ligands are active agents in drugs, and their binding can modulate the biological activity of the target protein, either inhibiting or enhancing its function. **Molecules**, in a broader sense, refer to any chemical structure consisting of

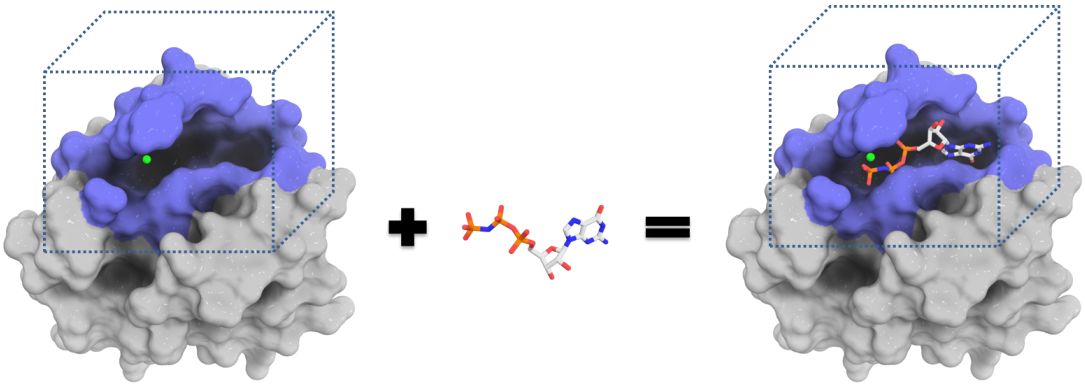


Figure 2.1: A ligand fitting in one of the binding site of a protein

two or more atoms bonded together. In the context of drug discovery, the term *molecule* is often used interchangeably with *ligand*, especially when discussing small organic compounds that may serve as potential drugs. These molecules can exhibit diverse chemical properties, which influence their interaction with target proteins. The desired behavior for a molecule is to suppress the pathological effect of the disease (protein). There are several ways to evaluate the "goodness" of a molecule for a specific protein, but the docking phase itself is out of the scope of this thesis. **Ligands are flexible**: they can adopt multiple spatial arrangements, known as **conformers**. In the nature was observed that molecules have unpredictable behaviors: they rotate around its chemical bonds, therefore adopting various conformations. The ability to adopt many conformations is important because the efficacy of a ligand depends on its ability to fit and interact optimally with the target site of a protein.

SMILES

The Simplified Molecular Input Line Entry System (SMILES) is a compact, line-based notation that encodes a molecule's connectivity and stereochemistry using alphanumeric characters [27]. SMILES strings include:

- atoms: Represented by their chemical symbols (C, N, O, etc.)
- bonds: Single bonds with - (atoms share one pair of electrons), double with =(atoms share two pair of electrons (four electrons)), triple with # (three pairs of electrons), represented with / for stereochemistry, aromatic with : (denotes delocalized π-electrons in a conjugated ring, that provides more stability to the molecule), and disconnected molecular fragments with the dot "."
- rings: Indicated by matching numbers to "close" the ring, e.g., C1CCCCC1 for cyclohexane (Figure 2.2): this ring contains six Carbon atoms. In the graph of the

molecule, there is one ring, and this motivates the "1" in the SMILES notation, which serves as the identifier for the ring. This system allows for the unique assignment of ring identifiers, making it possible to represent molecules with multiple rings without ambiguity. The ring in this case consists of six carbon atoms. This structural feature is reflected in the SMILES notation by enclosing the ring atoms and using numerical identifiers to indicate connectivity. In the case of cyclohexane, the SMILES representation is "C1CCCCC1," where each carbon is explicitly connected in a continuous cycle, and the "1" ensures that the first and last carbons are linked to close the ring.

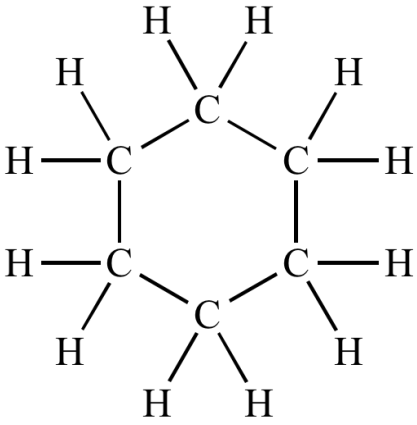


Figure 2.2: Cyclohexane (SMILES: C1CCCCC1) molecule with a ring of 6 Carbon atoms

- branches: Parentheses () indicate branching from the main chain. They indicate a separation from the main structure of the molecule.

This representation includes all the information needed for the identification of the molecules and its chemical properties, except for the hydrogen atoms (as we can see in Figure 2.2, the cyclohexane SMILES doesn't contains any information about hydrogen atoms, even though the ligand contains hydrogen atoms).

3D Coordinates

Historically, detailed three-dimensional coordinates of each atom in a molecule, derived from experimental (e.g., X-ray crystallography) or computational methods, have been used to study molecular conformation and interactions.

This representation is used in the docking phase and energy minimization. In docking we need position of atoms and their spatial arrangement to evaluate interactions behaviors with the protein. In energy minimization because the energy depends on conformation of

the atoms and their arrangement in space.

Proteins are large biomolecules that play crucial roles in virtually every biological process. They are composed of long chains of amino acids (that are known molecules), which fold into specific three-dimensional structures. They participate in virtually every biological process, serving as enzymes, receptors, and structural components.

A **target site**, often referred to as the *binding site*, is the specific region on the protein where the ligand binds. The binding of a ligand to the target site can induce conformational changes in the protein structure, either activating or inhibiting its biological function.

2.2. Force Field

Molecular mechanics (MM) treats atoms as particles that follow classical Newtonian physics, calculating interactions as a function of the nuclear positions only. The parameters for these interactions are partly derived from quantum mechanics calculations and partly from fitting experimental data; hence, **semi-empirical**. MM is based on a relatively simple model that defines the forces acting on biomolecules. These forces are results of different types of interactions, like the stretching of **bonds**, the opening/closing of **angles**, single bond **rotations**, and **non-bonded interactions** such as van der Waals forces and electrostatics. The force acting on each atom or group of atoms in the system is defined by these interactions, with deviations of bonds or angles from their equilibrium associated with energetic penalties.

At the core of Molecular Mechanics lies the **force field**, which establishes the forces acting on molecules. A force field is defined by two components: the **functional form** (which describes how energy depends on the atomic positions and motions) and the **parameters** associated with these functions. These parameters are typically derived from fitting data obtained from quantum chemical calculations on small, well-characterized molecules, as well as from experimental results. The choice of the functional form and fitting parameters determines both the accuracy of the force field and its computational efficiency.

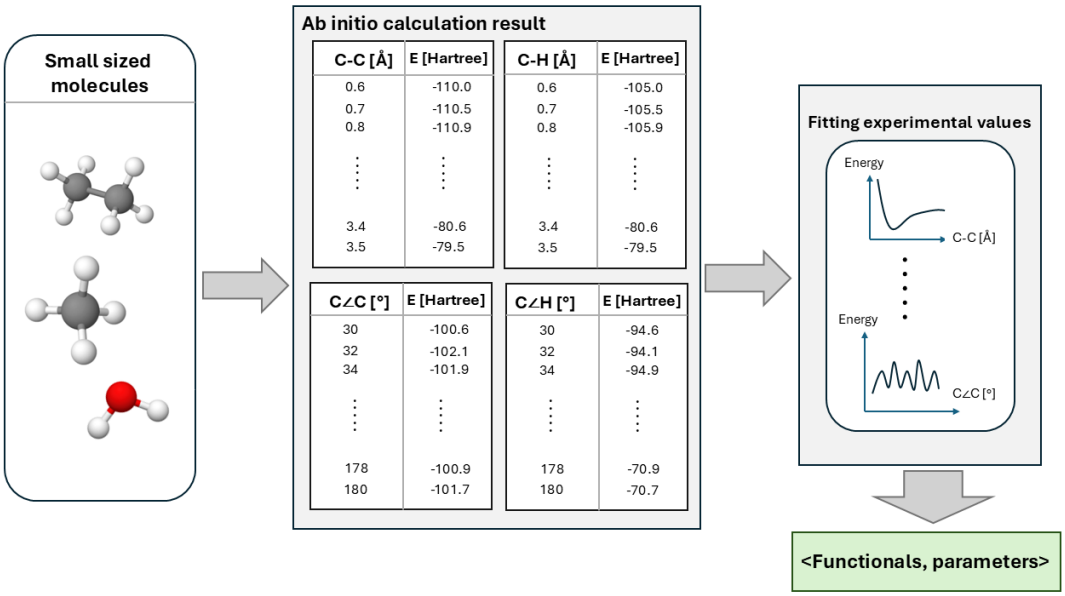


Figure 2.3: Typical procedure to build empirical force field

A key property of force field is their **transferability**: the same set of parameters can be used to model a series of related molecules, eliminating the need to develop new parameters for each molecule. This property is crucial if we want to use the force field to make predictions, applying a single set of parameters across different molecules.

The last key component in force field definition is the **atom type**. It is more than just the atomic number of an atom; it usually contains information about hybridization state and its local environment. For instance, is necessary to distinguish oxygen bond with a single bond, double bond, bound to hydrogen or in a ether group.

Force Fields are developed following the same schema shown in Figure 2.3; accurate quantum level simulations are performed with many small well studied molecules (like CH_3, H_2O, C_2H_6 ecc.). Extensive data are retrieved from those calculations, such as studying how energy changes with respect to elongating bonds, or by increasing systematically the angle between triplets of atoms. Once many data are retrieved, fitting phase of the energy with the distances and angles is performed with high statistical accuracies. The fitting can be carried out in different ways, influencing significantly how the force field will then be implemented. The two main approaches to fitting are **empirical** or **machine learning-based**, with the key difference being how the features are obtained.

In the empirical approach, the features are predefined, based on established human understanding of the physical system (e.g., the energy of a system composed of two atoms decreases as the distance between them increases). In contrast, with the machine learning-

based approach, features are typically learned by neural networks, which automatically identify patterns from data without explicit human-defined rules.

2.2.1. Empirical Force Fields

Empirical force fields are relatively easy to interpret because they are grounded in human knowledge, gathered in years of research in quantum chemistry and biology. An important concept to stress out is the *empirical* word: there is not a "correct" form for a force field. Instead, the choice of the mathematical modeling of the energy, is guided by the need of achieving a good balance between accuracy and computational efficiency. While a more complex and accurate functional form might provide more accurate results, it can be computationally expensive, limiting its practical use.

There is a wide variety of molecular empirical force fields, each built for different aims: quantum chemistry force fields aims to the most accurate possible energy description, modeling also electrons behaviors. In contrast, in biology, electrons are typically discarded, following the Born-Oppenheimer approximation: since the mass of the nuclei is much larger than the mass of the electrons, the nuclei generally move much slower than the electrons. Therefore, the motion of the nuclei are decoupled from the motion of the electron. Most widely used force fields in biology research are AMBER, developed for simulating proteins and nucleic acids and CHARMM, that is primary used for biomolecular systems like proteins, lipids and nucleic acids.

2.2.2. Merck Molecular Force Field

Merk Molecular Force Field (MMFF94) is an empirical force field that will be used in this thesis: is optimized for small molecules, making it ideal for pharmaceutical modeling, and is currently computationally efficient for ligand-based simulations. For this reason is explain in detail. Its parameters are finely tuned through both quantum mechanical calculations and experimental data, which enables it to deliver precise energy evaluations and geometries for molecules.

The MMFF94 energy expression can be written as:

$$E_{MMFF} = E_{BondStretching} + E_{AngleBending} + E_{StretchBend} + E_{OutOfPlane} + E_{Torsion} + E_{vdw} + E_{charged} \tag{2.1a}$$

How each component in the summation is evaluated will be explained in the next sections.

Bond Stretching interaction

The stretch bonded contribution model the energy term raised by bonded atoms in pairs that moves in frontal direction like a spring. Before delving in the mathematical formulation, we may figure out the interpretation of a bond interaction as a spring. The more the two atoms are moved in the same direction, the more the spring will push the two parts away, as a repulsion force. This model makes the two atoms not to stay in the same position, avoiding not plausible conformations. The potential energy curve for a typical bond has the form shown below:

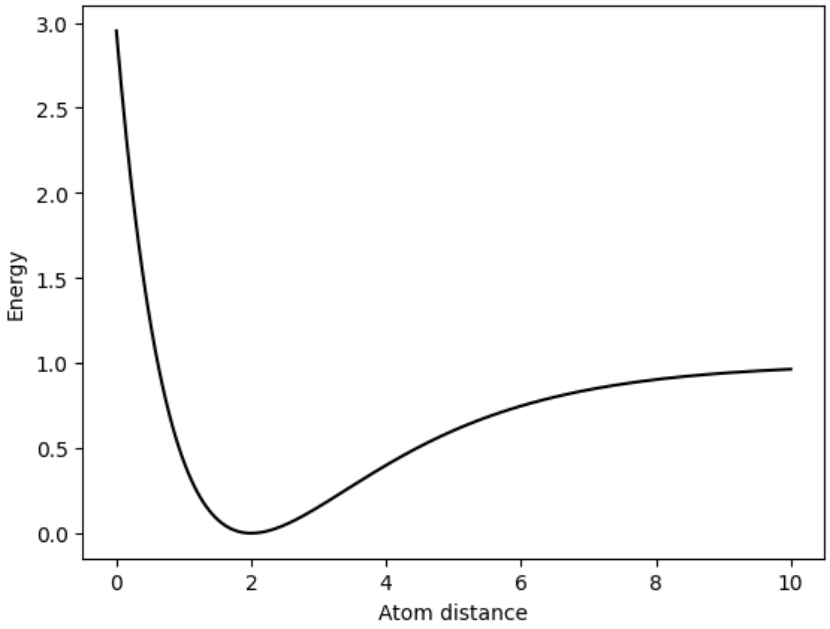


Figure 2.4: Morse potential

The Morse potential has the form:

$$V(l) = D_e \left(1 - \exp(-\alpha(l - l_0))\right)^2 \tag{2.2}$$

where D_e is the depth of the potential energy minimum, $\alpha = \omega\sqrt{\mu/2D_e}$ where μ is the reduced mass and ω is the frequency of the bond vibration. l_0 is the reference value of the bond.

The morse potential is not used in molecular mechanics force field because is not amenable to efficient computation, being an exponential expression. Moreover, Morse potential accounts for bond dissociation, which is important in ab-initio calculations, but in classical force fields bonds breaking is not modeled.

MMFF94 employs the quartic function, calculating the energy between atom i and j as:

$$EB_{ij} = 143.9325\frac{kb_{ij}}{2}\Delta r_{ij}^2(1 + cs\Delta r_{ij} + \frac{7}{12}cs^2\Delta r_{ij}^2) \tag{2.3}$$

where kb_{ij} is the force constant (md/Å), Δr_{ij} is the difference between length between the length of the current conformation of the molecule and reference bond lengths (that in the spring scenario is the "rest length"). This constant depends on the atoms types of the two atoms i and j. $cs = -2\text{Å}^{-1}$ is the *cubic stretch constant.*

Angle Bending

Angle bending potential energy is similar to the bond stretching. For this interaction, MMFF94 employs a cubic expansion:

$$EA_{ijk} = 0.043844\frac{ka_{ijk}}{2}\Delta\theta_{ijk}^2(1 + cb\Delta\theta_{ijk}) \tag{2.4}$$

where ka_{ijk} is the force constant (md Å / rad^2) and depends on the atom types of the triplets of atoms i, j, k, $\Delta\theta_{ijk}$ is the difference between current and reference angles, and cb is the *cubic bend constant.* Rather less energy is required to distort an angle away from equilibrium than to stretch or compress a bond, therefore, the force constants are proportionally smaller.

For linear or near-linear bond angles, MMFF94 uses another well-behaved form instead:

$$EA_{ijk} = 143.9325ka_{ijk}(1 + \cos\theta_{ijk}) \tag{2.5}$$

Stretch-Bend interactions

They describe how the energy of a molecule changes when the bond lengths (stretches) and bond angles (bends) are coupled or interact. Are used to more accurately model complex or flexible systems, where changes in bond lengths can influence bond angles and vice versa. MMFF94 employs the form:

$$EBA_{ijk} = 2.5121(kba_{ijk}\Delta r_{ij} + kba_{kji}\Delta r_{kj})\Delta\theta_{ijk} \tag{2.6}$$

where kba_{ijk} and kba_{kji} are force constants (md/rad) that couple the $i - j$ and $k - j$ stretches to the $i - j - k$ bend. They depends on the atom types of the two triplets of the quadruplets of atoms. Stretch-bend interactions are omitted when eq. (1.11) is used

for bond angles.

Out-Of-Plane bending interactions

Given a set of four atoms, the *Wilson angle* is the angle between the plane formed by $i - j - k$ atoms with the bond $j - l$:

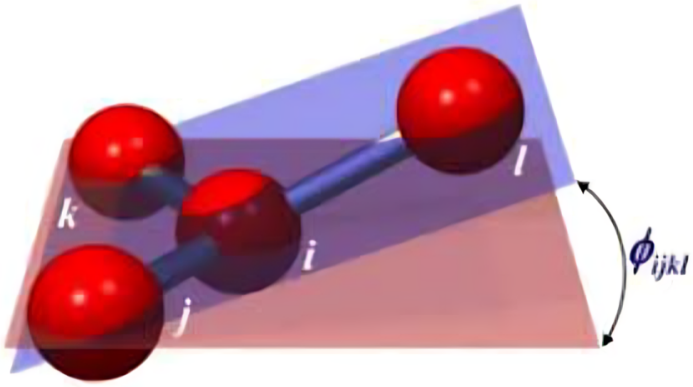


Figure 2.5: Wilson angle

A value of 0° corresponds to all four atoms being coplanar.

This interactions affect the energy by the contribution from the movement of an atom that is part of the planar structure in a direction out to the plane formed by the other atoms. In biological systems, planarity is crucial for the function of molecules. For example, amide bonds (like those found in proteins) typically favor a planar configuration, and out-of-plane bending can disrupt this structure. MMFF94 uses the form:

$$EOOP_{ijk;l} = 0.043844 \frac{koop_{ijk;l}}{2} \chi^2_{ijk;l} \tag{2.7}$$

where $koop_{ijk;l}$ is the force constant (md Å / rad^2) and $\chi_{ijk;l}$ is the Wilson angle. If this interaction is discarded, for some well-studied structures, like the *ketone*, the equilibirum structure would have the oxygen atom located out of the plane formed by the carbon atom and the two carbon atoms bonded to it. But experimentally, it is found out that the oxygen atom remains in the plane of the three carbon atoms. In order to achieve the desidered geometry, this term is necessary since penalize Out-Of-Plane bendings, forcing them to stay in the same plane.

Torsional interactions

Most of the variation in structure and relative energies is due to the complex interplay between torsional and non-bonded contributions. Given a set of four atoms, the *torsional* or *dihedral* angle is the angle between the two intersecting planes formed by the three pairs of bonded atoms. In order to fully describe the torsional characteristics of a molecule composed of four bonded atoms, it is necessary to consider two key angular parameters: the bond angle (θ) and the torsional or dihedral angle ϕ.

The bond angle θ is the angle formed between three consecutively bonded atoms. In Figure 2.6, this angle is defined between atoms 1, 2, and 3. It represents the deviation of the central atom (atom 2) from a perfectly linear arrangement. The dihedral angle ϕ, on the other hand, is a measure of the torsional rotation around the central bond. It is defined as the angle between the two planes formed by atoms (1-2-3) and (2-3-4). In other words, it describes how the two terminal atoms (1 and 4) are positioned relative to each other when rotating around the central bond between atoms 2 and 3. Changes in this torsional angle significantly affect molecular conformation and energy due to steric hindrance and non-bonded interactions. The energy profile associated with ϕ variations often leads to different conformational states, such as staggered configurations.

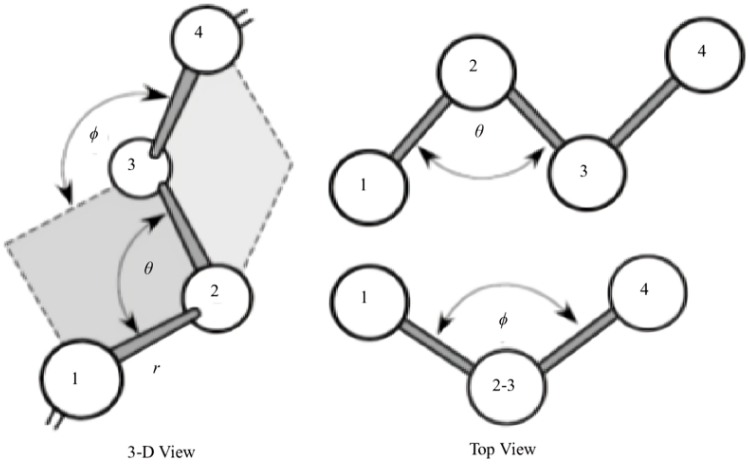


Figure 2.6: Torsional angle from "Effect of doped indium" by Muhaj Talib et. al.

A classical example of how energy changes with bond rotation is observed in ethane (C_2H_6), where the relationship between torsional strain and dihedral angles leads to the energy profile shown in Figure 2.7. The energy variation arises due to steric and electronic interactions between hydrogen atoms on adjacent carbon atoms as the molecule undergoes

rotation around the carbon-carbon single bond.

The graph displays the energy profile as a function of the torsional angle, where distinct conformations appear periodically. The energy minima correspond to the staggered conformations (dihedral angles of 0°, 120°, and 240°), which are the most stable due to minimized repulsions between hydrogen atoms. In these conformations, the hydrogen atoms on one carbon are positioned between the hydrogen atoms on the adjacent carbon, reducing electron cloud interactions. Conversely, the energy maxima correspond to the eclipsed conformations (dihedral angles of 60°, 180°, and 300°). In these cases, hydrogen atoms on adjacent carbons align directly with another one. This results in higher energy states, making eclipsed conformations less favorable compared to staggered ones. This repeating energy pattern follows a sinusoidal trend, highlighting the periodicity of torsional strain in ethane.

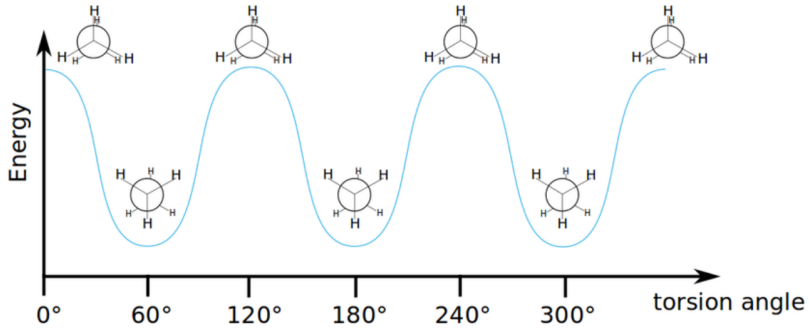


Figure 2.7: Torsional contribution energy to ethane

Torsional potentials are almost always expressed as a cosine series expansion due to the periodic energy values obtained. MMFF94 uses the form:

$$ET_{ijkl} = 0.5(V_1(1 + \cos(\phi)) + V_2(1 - \cos(2\phi)) + V_3(1 + \cos(3\phi))) \tag{2.8}$$

The bonded pairs are $i - j$, $k - j$ and $k - l$. The constants V_1,V_2,V_3 depend on the atom types.

Electrostatic interactions (non-bonded interactions)

Charge distribution in a molecule is represented by assigning fractional point charges throughout the molecule: a charge q_i of 0.75 C is assigned to atoms being 3 bonds far, while a charge of 1 C when atoms are 4 bonds away. These charged are designed to reproduce the electrostatic properties of the molecule.

MMFF94 uses the buffered Coulomb function:

$$EQ_{ij} = 332.0716 \frac{q_i q_j}{D(R_{ij} + \delta)} \tag{2.9}$$

where q_i and q_j are partial atomic charges and depends on atoms types, R_{ij} is the internuclear separation in Å, δ is the electrostatic buffering constant and D is the dielectric constant.

Van der Waals interactions (non-bonded interactions)

From the formulation of the electrostatic interaction, is immediate to notice that is a completely repulsive force. Electrostatic interactions cannot account for all non-bondend interactions in a system. Ideal gases have been deeply studied in the past, and was observed that the interaction energy between two atoms varies with the distance between them. At short distances, the atoms experience a repulsive force, while at greater distances, they are subject to an attractive force.

The **attractive force** arises due to dipoles formed by fluctuations in the electron clouds of atoms. A dipole in one atom induce a dipole in neighboring atom, leading to an attractive interaction.

On the other hand, the **repulsive contribution** has a quantum mechanical origin. According to the Pauli exclusion principle, no two electrons in the same system can have identical quantum numbers. As a result, when atoms come very close together, the reduced electron density in the region between the nuclei causes a repulsion between them, as the atomic orbitals begin to overlap and the system resists the compression of electron clouds.

Plot in Figure 2.8 shows the Van der Waals energy in relation to the distance between non bonded atoms. In abscissa the distance between atoms in 3 D space is represented, that is $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ where subscript i represent coordinates for atom i and j for the other, the positive region of the potential curve corresponds to the repulsive interaction, which occurs when atoms are very close together and experience strong repulsion. The negative region of the potential curve represents the attractive interaction, which dominates at intermediate distances between atoms. This attraction lowers the energy of the system, and it's the interaction we aim to maximize in order to minimize the overall energy of the system. In other words, the system reaches a more stable conformation when the attractive forces are stronger than the repulsive ones.

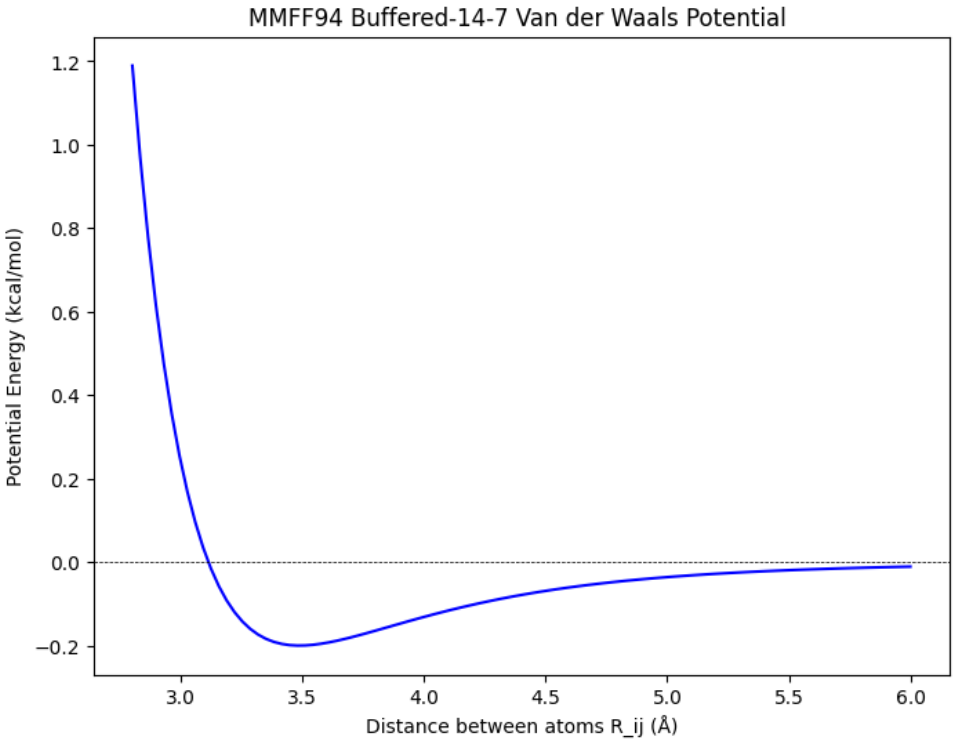


Figure 2.8: Van der Waals potential

Several functionals of this interactions are developed, but MMFF94 implements the *"Buffered-14-7"* form. Terminology indicating the 14-th and 7-th power dependence for the repulsive and attractive force. The term "buffered" indicates the inclusion of additional constants in the distance terms to smooth the transition between the attractive and repulsive forces.

The equation for the van der Waals interaction energy between two atoms i and j is given by:

$$Evdw_{ij} = \epsilon_{ij}\left(\frac{1.07R^*_{ij}}{R_{ij} + 0.07R^*_{ij}}\right)^7\left(\frac{1.12R^{*7}_{ij}}{R^7_{ij} + 0.12R^{*7}_{ij}} - 2\right) \tag{2.10}$$

where ϵ_{ij} is the depth of the potential well, and depends on both atoms' types, R_{ij} is the distance between atoms and R^*_{ij} is a reference distance related to the radii of the interacting atoms.

Van der Waals and electrostatic interactions between two atoms are considered whenever the two atoms are separated by at least three chemical bonds.

2.2.3. Machine Learning-based

Over the last decades, **machine learning (ML)** models have emerged in a vastness of scientific problems, including Molecular Mechanics. In this context, are used as an alternative approach to build force fields. ML-based force fields can learn the feature's of the molecules that affect the potential from datasets of quantum mechanical calculations or experimental measurements, without any prior chemical knowledge (that is present in *empirical* force fields).

Force fields like ESPALOMA [23] (Extensible Surrogate Potential for Atomistic Learning of Molecular Attributes) leverages graph neural networks (GNNs) to parameterize molecular interactions. Instead of relying on traditional heuristic-based atom typing, ESPALOMA directly learns atom-type embeddings from molecular graphs, enabling more flexible and automated force field assignment.

Their accuracy of the PES landscape, however, comes at the expanse of higher inference time. For this reason, Machine Learning Force Fields are not widely used in High-Performance Virtual Screening. They're main aim is to narrow the gap between the accuracy of ab initio methods and the efficiency of classical FFs. [24] Those force fields are instead chosen when accuracy is the main focus, like in quantum chemical calculations.

2.3. Energy minimization

In molecular mechanics, the system's **potential energy E** is a multidimensional function that depends on the positions (coordinates) of all atoms in the molecule. These positions can be denoted by $\{ r_i \}$ where each r_i is the coordinate vector of the i-th atom. Because the energy depends on multiple atomic coordinates, this function spans a high-dimensional space known as the **potential energy surface (PES)**. Conceptually, every point on the PES corresponds to a unique arrangement (conformation) of the atoms in the system, and the value of the function at each point is the total potential energy for that atomic arrangement.

For a molecule of N atoms, the PES is a function of $3N$ variables (the x, y, and z coordinates for each atom, minus any constraints such as rigid-body translational or rotational degrees of freedom). Graphically, it is often described as a multi-dimensional "landscape" with hills (high-energy regions) and valleys (low-energy regions).

Each of the term in Equation 2.1a contributes to the overall shape and ruggedness of the PES.

A **local minimum** on the PES is a point where the gradient (or forces) is zero, and small displacements of atoms in any direction lead to an increase in energy. However, a local minimum may or may not be the **global minimum**, which represents the absolute lowest energy state accessible on the PES. Due to the often rugged nature of the PES, many local minima can exist, separated by energy barriers.

At a local minimum, forces on each atom are balanced, implying a **stable** or **metastable** configuration. If no further decrease in energy is possible by small displacements, the system is said to be in a local minimum. The global minimum is the overall most stable structure, but finding it directly can be challenging because of the large number of possible conformations.

The objective of energy minimization is to find one of these minima on the PES—often a local minimum of interest—by systematically reducing the total potential energy. Mathematically, a minimum is characterized by requiring that the forces on each atoms vanish. This condition is expressed as:

$$\sum_{i \in \{atoms\}} \frac{\partial E(r_i)}{\partial r_i} = 0 \tag{2.11}$$

Because molecular systems can have many local minima, the minimization procedure typically finds a stable conformation near the starting structure rather than guaranteeing the global minimum.

Numerical optimization algorithms iteratively adjust the atomic coordinates so that the total energy decreases. Each iteration involves calculating the forces on each atom, updating positions to reduce these forces, and checking convergence criteria.

Common numerical algorithms include:

- **Steepest Descent**: Moves atoms in the direction of the negative gradient. Simple and robust but can converge slowly near the minimum.
- **Conjugate Gradient**: Similar to steepest descent, but uses previous gradient information, leading to fewer steps to convergence.
- **Quasi-Newton Methods**: Build up an approximation to the Hessian matrix (second derivatives), allowing for rapid convergence. Converge faster for medium-sized to large systems, since the second derivative information is used to understand the convexity of the PES.

Non-numerical or heuristic approaches (like Genetic Algorithm, Simulated Annealing,

Basin Hopping) don't rely on gradient information. Instead, they explore the PES more broadly, potentially overcome easily energy barriers. However, they do rely heavily on fine tuning of parameters, and typically need higher computational effort.

Quasi-Newton methods are widely used in energy minimization of small molecules. This section will describe both BFGS and its memory efficient variant L-BFGS, along with their mathematical formulations.

2.3.1. BFGS algorithm

The **BFGS** algorithm is a quasi-Newton method that is widely used for energy minimization due to its robustness and relatively fast convergence. The core idea of the BFGS method is to iteratively improve an approximation of the inverse of the Hessian matrix, which is used to predict the curvature of the PES. By doing so, the BFGS method provides a way to adjust atomic positions in a manner that converges rapidly to the minimum energy configuration.

The algorithm starts with an initial configuration of atomic coordinates x_0, and an initial approximation for the inverse Hessian, $H_0^{-1} = I$. At each iteration k the gradient of the potential energy with respect to atomic position is calculated:

$$\mathbf{g_k} = \nabla \mathbf{E}(\mathbf{x_k}) \tag{2.12}$$

where x_k are the atomic positions at iteration k, and g_k is the gradient (also referred to as the force $F = -\nabla E$).

The search direction p_k is determined by multiplying the inverse Hessian approximation by the negative gradient:

$$\mathbf{p_k} = -\mathbf{H_k^{-1}}\mathbf{g_k} \tag{2.13}$$

This condition guarantees that the algorithm moves in the direction that reduces the potential energy.

A line search is performed to determine the optimal step size, α_k, that minimizes the energy along the search direction p_k. The **Wolfe conditions** are used to ensure both sufficient decrease of the objective function and that the curvature of the PES is respected. Wolfe condition consists in:

- The **Armijo Condition** (Sufficient Decrease Condition) ensures that the energy

decreases sufficiently as the step size increase.

$$\mathbf{E}(\mathbf{x_{k+1}}) \leq \mathbf{E}(\mathbf{x_k}) + c_1 \alpha_k \nabla \mathbf{E}(\mathbf{x_k})^{\mathbf{T}} \mathbf{p_k} \tag{2.14}$$

This ensures that the energy is decreasing at a rate proportional to the step size.

- The **Curvature Condition** ensures that the slope of the energy function is reduced sufficiently in the direction of the search.

$$\nabla \mathbf{E}(\mathbf{x_{k+1}})^{\mathbf{T}} \mathbf{p_k} \geq c_2 \nabla \mathbf{E}(\mathbf{x_k})^{\mathbf{T}} \mathbf{p_k} \tag{2.15}$$

This condition guarantees that the step is not too large and that the curvature of the PES is respected.

The new atomic coordinates are then updated, making a step in the direction with that α_k amount:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k} \tag{2.16}$$

After each step, the inverse Hessian approximation is updated using the BFGS update rule. First, the difference between consecutive positions and gradients is calculated:

$$\mathbf{s_{k+1}} = \mathbf{x_{k+1}} - \mathbf{x_k}, \qquad \mathbf{y_{k+1}} = \mathbf{g_{k+1}} - \mathbf{g_k} \tag{2.17}$$

The inverse Hessian is finally updated:

$$\mathbf{H_{k+1}^{-1}} = \left(I - \frac{s_k y_k^T}{y_k^T s_k} \right) \mathbf{H_k^{-1}} \left(I - \frac{y_k s_k^T}{y_k^T s_k} \right) + \frac{s_k s_k^T}{y_k^T s_k} \tag{2.18}$$

This update ensures that the Hessian remains positive definite and that the algorithm converges toward a minimum.

The algorithm continues until the magnitude of the gradient g_k is below a threshold, meaning that a local minimum has been found. The BFGS algorithm provides quadratic convergence in practice. However, its main limitation is the storage and computational cost of maintaining and updating the Hessian, which scales as $O(n^2)$ for n atomic coordinates. For large systems, this can become expensive.

2.3.2. L-BFGS algorithm

The **Limited-memory BFGS (L-BFGS)** algorithm addresses the storage limitations of BFGS by approximating the inverse Hessian using only a limited number of recent updates. This makes L-BFGS more memory-efficient, especially for large systems, while retaining many of the benefits of the BFGS algorithm.

L-BFGS uses a compact representation of the inverse Hessian approximation by storing only a limited number of vector pairs (s_k, y_k) from recent iterations, instead of the full inverse Hessian.

In detail, the inverse Hessian H_k^{-1} is approximated using the stored (s, y) pairs from the previous m iterations, where m is a parameter that controls the memory usage. The search direction p_k is determined by applying a two-loop recursion that efficiently computes the approximate product of the inverse Hessian and the gradient. The two-loop recursion avoids the need to explicitly form or store the full Hessian matrix.

Memory and computational cost are reduced to $O(nm)$, where m is typically small compared to the number of atomic coordinates n.

2.3.3. Energy Minimization Software

This section is dedicated to a brief review of available energy minimization software used in research and industry. These applications are typically complex and comes in a modular organization. Each module responsible of various chemical relevant tasks, indeed, no software performs solely energy minimization.

GROMACS [25] is one of the most used molecular simulation package, and its modules are, apart from energy minimization, MD simulations for proteins, lipids and nucleic acids, free energy calculations and normal mode analysis.

The main focus of this thesis is on the module that minimize energy of molecules from their SMILES representation. Those modules are mainly a combination of Force Fields (introduced previously in 1.3.2) and optimization algorithms (1.3.4).

Efforts will be dedicated on LiGen software since the aim of the thesis is to increase the throughput of this application.

2.4. Virtual Screening

Virtual Screening (VS) is a set of computational techniques used to predict potentially bioactive compounds from large databases of chemical structures.

In the earlier days of drug discovery, laboratories relied on wet-lab screening, testing thousands of compounds against biological targets in a phisical way. The dream situation would be to test the most as possible of the vast chemical space of molecules, that is almost not finite. Probing such a vast amount of molecules against proteins, became prohibitively expensive and time-consuming [22].

The advent of high-performance computing (HPC) and improved computational chemistry software has enabled the systematic assessment of large compound libraries in silico: that means testing compounds against targets by simulating their interactions. These advances have improved the efficiency of discovering potential compounds, saving resources, and guiding experimental efforts more effectively.

VS helps researchers prioritize which molecules in a chemical library should undergo in vitro or in vivo testing. This significantly reduces the time and cost associated with exhaustive experimental screening of vast compound collections.

Modern virtual screening campaigns often process billions of molecules [11]. The largest structure-based virtual screening experiment (EXSCALATE: EXaSCale smArt pLatform Against paThogE) [9] was carried out in the campaign against SARS-CoV-2, and consisted in more than 70 billion ligands. It showed that one of the main challenges was to generate the chemical knowledge of the virtual screening phase, that is the preprocessing phase of the ligands.

2.4.1. LiGen

LiGen is a molecular docking application designed to run on High-Performance Computers. It comes in a pipelined-modular fashion with the purpose of Virtual Screening. A module, called *ligen-pre* reads molecules in SMILES format and find its stable conformations. Another module, reads proteins, finding pockets, where the ligands will be placed in order to score the docking pose. After the preprocessing of ligands and the pockets identification of proteins are found, the actual docking occurs, ranking the top molecules that will be tested in lab. Since the docking stage is subsequent to the preparation of ligands, we will focus on describing the ligand preparation module of the software.

To better understand the inner complexity of ligen-pre, an high level, simplified workflow

is shown below.

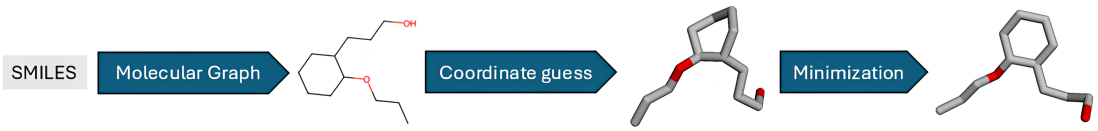


Figure 2.9: LiGen-pre workflow

Once the molecule in SMILES format is read from input (typically a `.smi` file), the parser proceeds by splitting and interpreting each relevant token according to the SMILES grammar. Using a parsing library (PEGTL) to build a *parse tree*, each node represents a fragment of the SMILES expression (atoms, ring closures, bonds, branching points, etc.). This parse tree is visited to construct the actual molecular representation: a graph whose vertices are atoms and edges are bonds.

The process follow this steps:

1. **Tokenization and Grammar Matching:** The input SMILES is read character by character, matching tokens such as atoms (e.g. `C`, `N`, `O`), branching symbols (parentheses), bond symbols (`-`, `=`,), ring specifiers (`1`, `2`, ...), and so forth. The grammar rules ensures that the input SMILES is a valid molecule.

2. **Parse Tree Construction:** Each token (or group of tokens, like bracket atoms `[NH3+]` or aromatic symbols `c`) becomes a node in the parse tree. This tree is hierarchical, capturing nested branches as children and storing relevant atom/bond information (charge, isotopes, stereochemistry, etc.).

3. **Graph Assembly:** For every node visited in the tree, a vertex in the graph is created. Then, it interprets bonds to connect atom vertices. For instance, when a ring closure digit (e.g. `1`) is encountered a second time, it "closes" the ring by connecting the current atom to the atom saved at the same ring index.

When building the molecule graph, also atom typing must be performed. It consists in looking up MMFF94 atom tables. This phase is handled by a sub-module called *ligen-dict* (dictionary). This stage is relevant for chemical purposes; in order to evaluate energy, the force field must be able to differentiate between different atoms *"flavors"* .

Once the connectivity graph with atom types information is built, atoms must be placed in the 3D space so that is chemically reasonable—i.e., that respects bond lengths, angles, ring shapes, chirality, and so on. For this scope, it takes each bond in the graph and assigns it an approximate distance derived from the MMFF94 parameters. The coordinate

assignment is an iterative approach: it builds both lower and upper distance bounds for every pair of atoms, so that any subsequent adjustments doesn't makes atom's distance too close or far away. These bounds reflect two-atom interactions (bond stretches), three-atom angles (where the distance between the outer atoms must accommodate a specific bond angle at the central atom) to ring topologies and dihedral constraints.

Once local constraints have been established, it picks a reference point in the graph (using a breadth-first traversal) and then assign coordinates to new visited atom. Initially, the positions are guided by the MMFF94 distances and angles. However, multiple iterations are performed to guarantee a coherent but flexible molecule. The program checks, pair by pair, whether the actual distance between two atoms violates the established distance bounds. If an atom pair is found to be too close or too far, both atoms are taken in opposite directions to bring them within the correct distance window. This step is repeated until the geometry better respects all the constraints.

During these corrections, ring systems are periodically unfolded and flattened: it takes a ring and attempts to place its atoms in a plane, and reorients substituents in a way that avoids strained or highly distorted geometries. Any chiral centers are then inspected to confirm that their substituents match the intended clockwise or counterclockwise arrangement; if they become inverted, this part of the molecule is mirrored to restore the chirality.

Once the coordinates of the molecule are assigned, it then proceed with the energy optimization through the L-BFGS algorithm.

2.5. HPC

The computational demands of molecular modeling (as described in Section 2.2 and by the mathematical formulation in 1.3) particularly in the context of virtual screening and energy minimization of vast molecular datasets, necessitate the utilization of High-Performance Computing (HPC) resources. This section discusses the main concepts of Central Process Units (CPUs), Graphics Processing Units (GPUs), GP-GPUs and High-Performance Infrastructures (target architectures) since are required to understand choices made in the methodology.

The key theoretical concept to grasp is **parallelism**. It can be implemented in several ways. Processors utilizes pipelines to overlap execution of instructions and data movement. This is typically known as **Instruction-Level-Parallelism (ILP)**. CPUs detect parallelism in real time or rely on compile-time analysis to schedule operations. This

technique has improved single-processor performance but this strategy showed limitations both physical (in hardware) and conceptually, leading industry efforts shifts to multicore solutions.

Data-Level Parallelism (DLP) focuses on applying a single instruction to multiple data elements simultaneously. In this context, the primary focus is on **SIMD** (Single Instruction Multiple Data). The main application of this parallelism can be found in GPUs, where multiple parallel processors (Streaming Multiprocessors) execute a large number of lightweight threads efficiently.

However, while some architectures enforce strict lockstep execution, NVIDIA GPUs utilize **SIMT** (Single Instruction Multiple Threads), which allows for more flexible execution, where individual threads within a warp can diverge. On the other hand, AMD GPUs follow a wavefront-based execution model, which is closer to traditional SIMD but still provides some flexibility in execution. Therefore, a GPU is not a single SIMD processor but rather an array of SIMD units with varying execution models depending on the architecture.

More precisely, a GPU is a MIMD (Multiple Instruction Multiple Data) built from SIMD processors (SMs). Each processor has a small local memory, while all processors share a larger memory module.

Thread-Level Parallelism (TLP) tackles the limits of single-processor by running multiple threads. As multicore systems became the standard, designs began emphasizing how to distribute tasks and balance memory access. In smaller-scale symmetric multiprocessors, each processor shares the same memory resources, whereas larger designs feature distributed memory banks to accommodate higher bandwidth demands. These distributed architectures often require additional coordination, as access times differ depending on where the data resides.

GPU

Since the main focus of this thesis will be on parallelization on NVIDIA GPUs, in this chapter an explanation of GPU architectures is provided. NVIDIA GPUs follow a design that prioritizes massive parallelism, high throughput, and efficient use of on-chip resources. Rather than depending on powerful cores, they are built from multiple Streaming Multiprocessors (SMs), each containing lightweight execution lanes that operate in lockstep on groups of threads. This structure allows thousands of threads to run concurrently and hide memory latency by switching among **warps** (sets of threads that share the same instruction sequence but may differ in data (SIMD)) within each SM. A warp is considered

the smallest unit in the parallelism.

A single GPU can be seen as a MIMD, since each SM can independently execute its instructions; however, inside each SM, individual instructions are typically SIMD, with the hardware lanes synchronized at the warp level. Each SM is fed by an hardware scheduler that issues instructions to warps. When one warp must wait on a memory operation or a long-latency instruction, the scheduler will switch to another warp that is ready to proceed, keeping arithmetic pipelines active (ILP). This rapid context switching and overlap of execution is a key to achieving high throughput in GPU computations. Keeping each threads in the warp executing the same instruction is fundamental to keep the expected SIMD parallelism implemented in hardware.

Parallelization is organized at the software level by decomposing work into thread **blocks** and **grids**. Each block is assigned to a SM, where it can occupy some subset or all of the SM's warp slots, depending on the number of threads and the availability of hardware resources such as registers and on-chip shared memory. Each block, in this way, can be scheduled to any available multiprocessor within a GPU, providing *automatic scalability* [17].

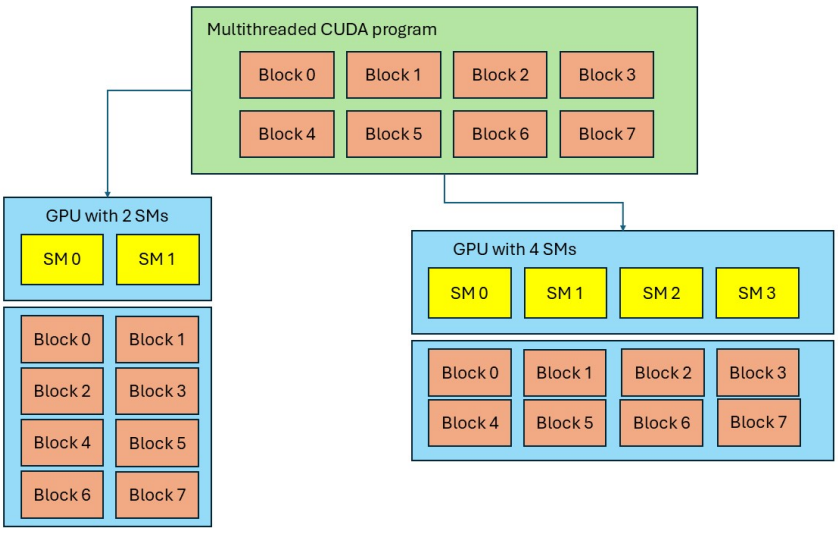


Figure 2.10: Automatic scalability [17]

Work is decomposed into thread blocks and grids, with each block assigned to a (SM). When a GPU executes a kernel, it schedules blocks to available SMs in **waves**. A wave refers to the set of blocks that can be executed concurrently across all SMs. For example, if a GPU has 20 SMs and each SM can handle 4 blocks simultaneously, the first wave would consist of 80 blocks. Once these blocks complete execution, the next wave of blocks

is scheduled. The concept of automatic scalability in CUDA programming is illustrated in Figure 2.10, which depicts a multithreaded CUDA program with multiple thread blocks mapped onto different GPU architectures with varying numbers of (SMs). Automatic scalability refers to the ability of a CUDA program to adapt dynamically to the available hardware resources without requiring any modifications to the program itself. This characteristic makes CUDA an efficient framework for parallel computing across a wide range of GPU architectures. In Figure 2.10, the multithreaded CUDA program consists of eight thread blocks, labeled from Block 0 to Block 7. These blocks represent independent units of work that can be assigned to SMs for execution. When this program is executed on a GPU with only two SMs, as shown in the lower-left section of the diagram, the blocks are distributed accordingly. Since there are only two SMs, they must process multiple blocks in a sequential manner. For example, SM 0 may start executing Blocks 0 and 2 while SM 1 handles Blocks 1 and 3. Once these complete, the next batch of blocks (4, 5, 6, and 7) will be scheduled on the available SMs, ensuring that all blocks are executed. However, when the same CUDA program is executed on a GPU with four SMs, as shown in the lower-right section of the image, the distribution of thread blocks changes automatically. In this scenario, the presence of additional SMs allows for greater parallelism, meaning that more blocks can be executed simultaneously. Each of the four SMs can handle a block at a time, allowing Blocks 0 through 3 to execute in parallel. Once they complete, Blocks 4 through 7 can be assigned to the now-available SMs, reducing the overall execution time compared to the two-SM configuration. This allow the programmer to not modify the code when moving from a GPU with two SMs to one with four SMs. Instead, the CUDA runtime system distributes the thread blocks across the available hardware, optimizing execution based on the GPU architecture. If the program were run on an even more powerful GPU with eight or more SMs, each block could potentially be assigned to a separate SM, leading to even greater efficiency. The **tailing effect** occurs when the total number of blocks is not evenly divisible by the number of blocks that can be executed in a single wave. For instance, if there are 85 blocks in total, the first wave would execute 80 blocks, and the remaining 5 blocks would form a smaller, final wave. This tailing wave can lead to under utilization of GPU's resources, as some SMs may remain idle while waiting for the final blocks to complete. At the same time, for some tasks, this effect is inevitable, simply because all elements must be computed, and not always is possible to balance a perfect mapping of blocks and computation parallelism.

Threads within a block can share data without fetching from the slower off-chip memory (in CUDA terminology **global memory**). Globally, all SMs are connected to large on-board GPU memory, with an L2 cache shared across the entire device that sits between

the SMs and off-chip memory. In addition to this L2 cache, each SM has a smaller L1 cache to keep frequently accessed data close to the processing lanes.

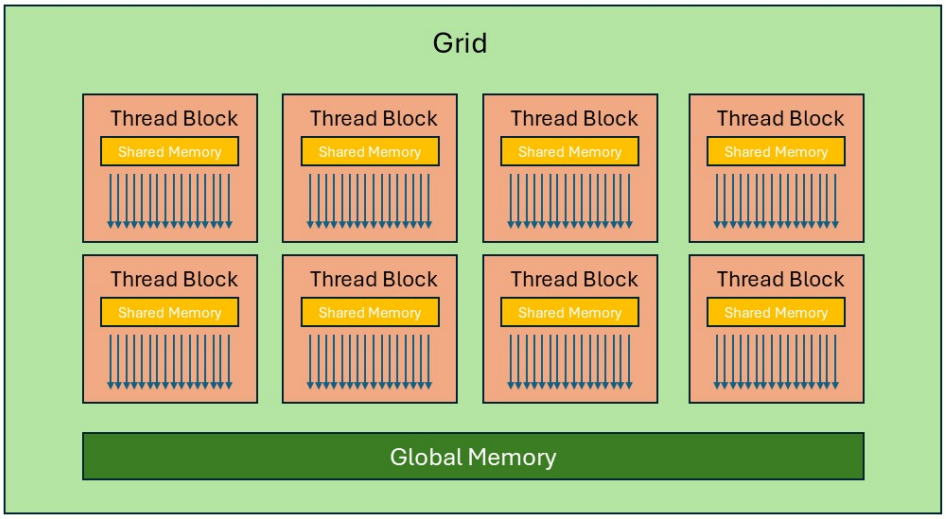


Figure 2.11: CUDA hierarchy
[17]

Figure 2.11 represents the CUDA hierarchy of components, where a **Grid** encompasses multiple **Thread Blocks**, each of which contains a set of **Threads**. Every thread block has its own **Shared Memory**, which is accessible only by the threads within that block, while all thread blocks share access to a larger **Global Memory**. This structure enables parallel computing by allowing multiple threads to execute concurrently while efficiently managing memory access at different levels. The efficiency of memory access patterns is critical for achieving high performance on GPUs. **Coalesced accesses** refer to the ability of threads within a warp to access consecutive memory locations in a single, consolidated transaction. This is particularly important when accessing **global memory**, as uncoalesced accesses can lead to multiple memory transactions, significantly increasing latency and reducing throughput.

In the CUDA hierarchy, global memory is the largest but slowest memory space, and it is shared across all SMs. To optimize memory bandwidth, threads within a warp should access memory in a **coalesced** manner. For example, if Thread 0 accesses memory location A, Thread 1 accesses A+1, Thread 2 accesses A+2, and so on, the GPU can combine these accesses into a single memory transaction. Conversely, if threads access

scattered or non-consecutive memory locations, the GPU may need to perform multiple transactions, leading to inefficiencies.

Register allocation also plays an important role in GPU performance. Each SM provides a pool of registers that are distributed among all threads. The more registers each thread uses, the fewer total threads can be allocated on that SM. **Occupancy** is the proportion of thread warps loaded on an SM.

CUDA developer guide discourage the usage of low amount of threads per block since those will limits occupancy. However, a common fallacy is to consider occupancy as a metric of utilization; it is not, is only one of the contributing factors. The reason of the CUDA guideline choice is that by running more threads per block the latency can be hidden. However, was also showed that maximizing occupancy, you may lose performance [10]. One reason is that, in GPU, we can use both ILP and TLP, hence, latency can be hidden also using more registers and not only increasing threads per block.

Since registers of the SM must be given to threads, having more threads per block would reduce the amount of registers per thread. At the same time in Figure 2.12, only registers are fast enough to get the peak, not even the shared memory, being almost 8x times slower than registers.

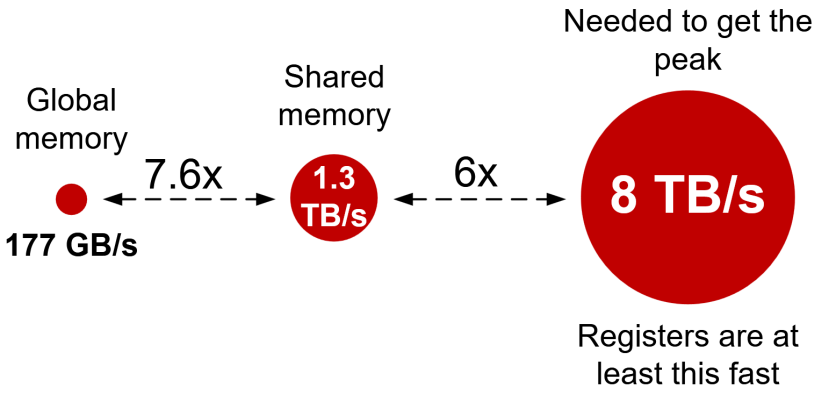


Figure 2.12: Must use registers to run close to the peak, this require many registers, that translates in lower occupancy. [10]

This is typically accomplished by computing multiple outputs per thread.

3 | Proposed Method

This chapter explains the strategy adopted to increase the throughput of LiGen, a molecular expansion application. The starting point is a serial executable, part of the LiGen ecosystem, which is first analyzed inferring the complexity of the problem using both prior domain knowledge and profiling in order to find optimization opportunities. Optimizations are then implemented only after bottleneck identification or data-driven observation of performance indicators. Parallelization both on CPU an GPU is the main techniques implemented. In Section 3.1 the LiGen-pre module's functions complexity is analyzed. After having established where the bottleneck is, in Section 3.2 we explain the CPU parallelization choice, motivating the idea behind and then implementation details. Finally, being LiGen target infrastructure are equipped with HPC-grade GPUs, in Section 3.3, strategies of the porting are shown and analyzed, always with motivations of any chosen optimization.

3.1. CPU Optimization

The original LiGen-pre module consisted in four main stages in the computation: the reading of SMILES from file-system, the building of the molecular graph using grammar SMILES parser, the assignment of coordinates to atoms. and finally the L-BFGS optimization with the objective function as the energy of the molecule, modeled with MMFF94.

Before any optimization, computational hotspot identification is necessary to understand which stage needs optimization or parallelization if feasible. To spot those computational hotspots and target specific functions to optimize, application profiling is performed on the application.

3.1.1. Latency parallelization

The profiling results show that while the minimization algorithm contributes the most to the overall execution time, the algorithm itself (LBFGS function) does not have a heavy

computational load, being its exclusive time is quite low, accounting for only 2.19% of the total time. A significant portion of the time is instead spent on updating and evaluating the energy at each optimization step, which is integral to the algorithm. In fact, all the energy components defined in Eq. (1.7a) are calculated at least once for each step of the optimization, but potentially up to 80 times (depending on the line-search hyper-parameters). This is evident also in the profiling from the fact that the calculation of the Van der Waals energy alone takes up 46.47% of the total time exclusively. Due to the nature of the Van der Waals interactions, which are non-bonded, the complexity of the calculation scales as $O(n^2)$ where n is the number of atoms.

Therefore, the I/O part of the computation is paltry, the bottleneck of the application is the energy computation. For this reason, it will be the target of optimizations. At the same time, also the workload to compute initial guesses of the coordinates is not negligible.

In light of this, we first aimed to reduce the load of the energy evaluation by computing each energy term of the MMFF94 in parallel using OpenMP [18]. Obtaining close to 1 speedup factor we tried to go in parallel over interactions for the non bonded energy contribution that showed to occupy more than half of the time. However, both strategies obtained poor speedup factor because ligands are small molecules and trying to optimize the single ligand was proven to not be a parallelization strategy suitable for extreme-scale ligand expansion.

3.1.2. Throughput parallelization

The end-to-end computation from SMILES to low-energy conformation is **embarrassingly parallel** from multiple perspectives. **From a molecule-wise point of view**, each molecule can be processed end-to-end independently from all the others. Ideally, all molecules minimum energy conformations can be generated independently from others.

From a stage-wise point of view (where a stage is either the reading of a molecule from file, the coordinate assignment or the minimization), each stage of a molecule can be carried out independently from the stage of others, for any stage of the process. Since the computation of one molecule does not affect the others, the process can be efficiently parallelized. When a thread is doing energy minimization of a molecule m_i, another thread can do the coordinate generation of another molecule m_j for any molecule in the dataset. This is true also for the same stage of the computation, not necessarily for different stages: when a thread is doing energy minimization of m_i, another thread can do energy minimization of m_j as well, independently.

In contrast, the original implementation is sequential with respect to molecule and stages. Each molecule must wait the previous one's completion before its processing. For this reason, the computation has been parallelized using **libdpipe** (a library that provides interfaces to implement pipelined softwares). It uses *stl threads* (Standard Template Library threads) to manage parallelism between stages of the pipeline, and uses thread-safe, finite queues with a **FIFO** (First In, First Out) policy to handle contents of each stages. At least one thread must be allocated for each stage; when we allocate one thread per stage, we're not falling back to sequential program, since still, each stage will be executed in parallel with respect to others.

Each stage processes elements from its input queue and, after performing the computation, pushes the results into the output queue, which is subsequently handled by the next stage in the pipeline.

Figure 2.1 shows the pipeline implementation with the role and content of each stage. Each process is represented by a circle, and the color coding reflects the object that the process is handling and the queue's content. In light sky blue we represent SMILES, in darker sky blue the molecule's initial guess, in pink buckets of initial guessed coordinates, and in green buckets of minimized molecules. t_p, t_b, t_m are hyperparameters of the pipeline; indicating the threads of respectively the guesser stage, the bucketizer and minimizer.

Each pipeline stage is now explained in detail.

1. **Reader**: read the input file containing SMILES string. This operation is sequential, therefore it runs on a single thread. It pushes SMILES read from file into the queue of the next stage.

2. **Guesser**: once a SMILES is received, it generates an initial guess for the 3D coordinates of the received molecule. This task can be parallelized, since each molecule's initial guess is independent from others. After generating the initial guess, this stage pushes the results into the bucketizer's queue.

3. **Bucketizer**: has the role of grouping initial guesses received from previous stage into *buckets* of fixed dimensions to enable batch processing of ligands.

4. **Minimizer**: once it receives a batch, it finds the most favorable energy conformation of every molecules in it. Once the batch is minimized, ligands of that batch are ready to be used for Virtual Screening.

Was considered the approach of processing ligands inside the bucket in parallel (*intra-bucket* parallelism), using OpenMP. However, this needed to allocate more threads inside each single minimizer. Allocating more minimizer with more threads saturated the

available core machine with little speedup factor due to context switches of cores. For this reason, *bucket parallelism* (with ligands inside the bucket performed serially) showed higher throughput than intra-bucket parallelism approach.

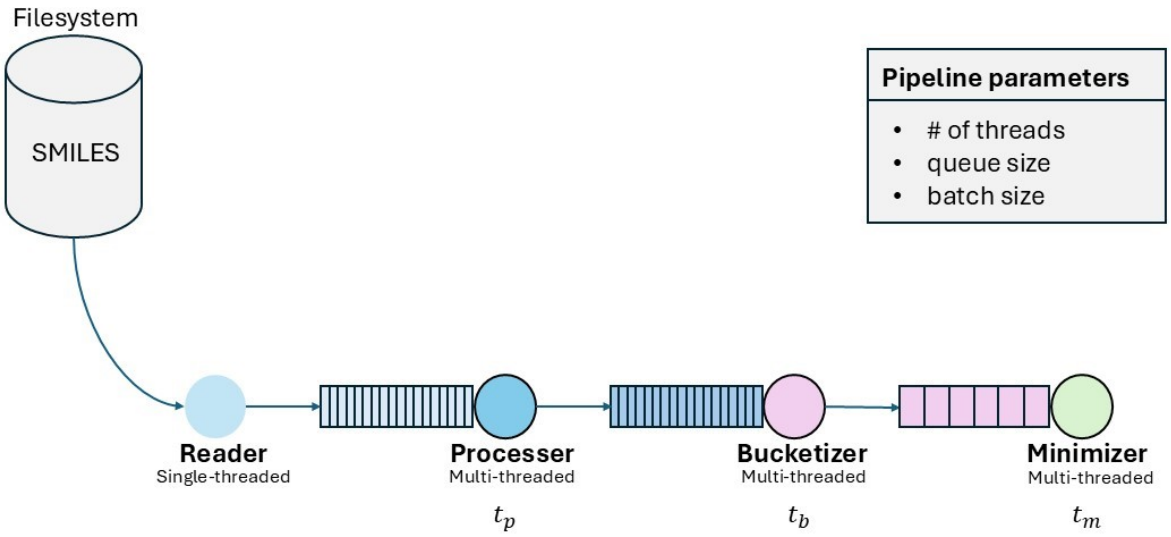


Figure 3.1: Ligen stages parallelized, organizing computations in a pipelined architecture.

The organized structure of this pipeline not only allows for the parallelization of each computational stage, but also provides a straight forward mechanism for **bottleneck detection**.

Using this pipeline approach, the application throughput is ideally the same of the throughput of the slowest process (if we don't consider initialization and termination of the pipeline, and passing considering passing of information between stage paltry).

By monitoring the queues size, we can identify bottlenecks as they occur. When a queue fills up, it indicates that the next stage is the slowest in the pipeline, creating a congestion and back-pressure effects. For instance in Figure 3.1 if the minimizer is the bottleneck, its queue will be the first one getting full, followed by the bucketizer that would push buckets into the minimizer it the latter queue was not full.

Implementing the bucketizer so that can be eventually parallelized resulted challenging because the threads running the bucketizer must manage their private queues before pushing data to the next stage, since its role is to accumulate a determined number of molecules before pushing. One considered approach has been to incorporate the bucketizer into the reader stage, moving only batches of molecules in the whole pipeline. This approach, however, doesn't allow eventual parallelization of the bucketizer, since a single thread (the reader) is managing the organization of molecules in this case.

The final bucketizer implementation consisted in a static queue shared by every potential thread of the bucketizer stage. With the implemented strategy, there is the advantage that multiple threads potentially operates on the same queue. We'll later see that computational demand of bucketizer is paltry compared to other stages, so, the parallelization of the bucketizer is not crucial in this case.

The pipeline is tested with 6000 randomly sampled ligands from the *EXSCALATE commercial dataset*. The procedure of building of the dataset is the same discussed in the previous section. The batch size is fixed to 100 molecules. Each queue can contain up to 200 molecules. For now, we configured $t_p, t_b, t_m = 1$. Below the stage's queue size evolution, representing the amount of ligands on queues of the stage (identified by colors). Every second the queues sizes are logged. The pipeline is configured with one thread per stage.

Allocating threads parsimoniously is crucial to implement an efficient parallelization. Moreover, allocating all threads to minimizer does not implies expected linear speedup. Even if the minimizer is the bottleneck of the pipeline, the more threads are allocated to the bottleneck stage, the more we expect that the bottleneck stage changes, with another one taking its place. Tuning the pipeline configuration is crucial to increase performances of the application, allocating threads only to stages that are strictly necessary.

In order to fine tune the threads amount for each stage of the pipeline an iterative and systematic approach was followed. First we executed the pipeline with one thread for each stage. Then the bottleneck is identified and more threads are allocated to that stages, so that multiple threads run the operation that resulted the most complex. When threads are allocated, it may happen that a new stage becomes the bottleneck. For this reason an iterative procedure was performed to find the optimal configuration of the pipeline.

3.2. GPU parallelization

Only the pipeline approach showed good performances in terms of throughput, however the single bucket is currently handled in serial in CPU, as we asserted that if also *intra-bucket* parallelization is implemented using CPU-thread parallelism no significant performance are gained. Moreover, target HPC architectures have also GPUs available on top of CPU cores, for more intensive workloads. Consequently, we decided to offload the minimization algorithm for bucket of molecules to GPUs. With respect to the state of the art that used to offload single molecule minimization algorithms or energy evaluation only to device, this is a novel approach of porting this type of batches calculation to device. The proposed method wants to use CPU parallelism for coarser level paralleliza-

tion (between buckets) and GPU parallelism for finer grain (ligands inside the bucket and ligand-wise too). If more than one GPU would be available, also bucket parallelization would be implemented in parallel but this scenario was not analyzed in this work.

Since this chapter shows the implementation on GPU of the minimization part of the application, it is crucial to delve more in depth about the L-BFGS algorithm implementation. For this reason, we show the pseudo-code of the whole minimization phase by an implementation point of view with respect to Chapter 2.

With the given pseudo-code, we stress out the computational load of the minimization stage showed by results of profiling in the previous section. The key aspect to keep in mind while reading this pseudo-code is that calculation of the gradient of a new conformation is not a cheap operation. Indeed, every time the coordinate vector x is modified and then g vector is used, we are calculating all energy contributions of equation 2.1a. This highlights and motivates the high percentages of time taken in profling by the non-bonded interactions. Indeed, most of the computational workload of the L-BFGS consists in the evaluation of the objective function and (especially) its gradient [13], rather than the actual algorithmic details of implementation.

In line 8 of Algorithm 3.1 (L-BFGS) there is the call to the line-searches; first the More-Thuente and if Wolfe conditions are not satisfied fall back to calling back-tracking line search. Both of the functions modify iteratively, until either Wolfe conditions are satisfied or *max_iterations* are performed, a constant α that is the coefficient of the direction's vector evaluated in Algorithm 3.2. For each candidate coordinates at each line search iterations, at **line 8 of the Algorithm 3.3** or similarly, at **line 6 of the Algorithm 3.4**, gradients and force evaluation is performed. Then gradient and vectors are used to build the Hessian matrix (from line 9 to line 13), which is crucial to estimate the most accurate as possible, in order to find accurate directions of minimization for the next stage, as illustrated in Algorithm 3.2 the importance of accurate Hessian matrix evaluation.

Algorithm 3.1 L-BFGS Optimization

Input:
- Initial guess $\mathbf{x}_0 \in \mathbb{R}^n$
- Gradient tolerance ε_g, function tolerance ε_f
- Maximum iterations $k_{\max}$
- History depth m

Initialize:
- $k \leftarrow 0$, history queue $\mathcal{H} \leftarrow \emptyset$
- Initial inverse Hessian approximation $H_0 = I$
- Compute initial gradient $\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$

while $\neg$ converged **and** $k < k_{\max}$ **do**
  Compute search direction:
  $\mathbf{p}_k \leftarrow \text{TwoLoopRecursion}(\mathbf{g}_k, \mathcal{H}, H_0)$ ▷ Algorithm 3.2
  Line search:
  Find $\alpha_k > 0$ satisfying Wolfe conditions: try More-Thuente first, fallback to backtracking
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$ (Update position)
  $\mathbf{g}_{k+1} \leftarrow \nabla f(\mathbf{x}_{k+1})$ (Update gradient)
  $\mathbf{s}_k \leftarrow \mathbf{x}_{k+1} - \mathbf{x}_k$ (Update history)
  $\mathbf{y}_k \leftarrow \mathbf{g}_{k+1} - \mathbf{g}_k$
  $\mathcal{H}$.push$(\mathbf{s}_k, \mathbf{y}_k)$
  if $|\mathcal{H}| > m$ **then**
    $\mathcal{H}$.pop_oldest()
  end if
  Check convergence:
  if $\dfrac{\|\mathbf{g}_{k+1}\|}{\max(1, \|\mathbf{x}_{k+1}\|)} < \varepsilon_g$ **and** $\dfrac{|f_k - f_{k+1}|}{\max(1, |f_k|, |f_{k+1}|)} < \varepsilon_f$ **then**
    converged $\leftarrow$ **true**
  end if
  $k \leftarrow k + 1$
end while
Return: $\mathbf{x}_k$ (Optimized coordinates)

Algorithm 3.2 Two-Loop Recursion for Search Direction

Require: Current gradient $\mathbf{g}$, history $\mathcal{H}$, initial Hessian H_0

$\mathbf{q} \leftarrow -\mathbf{g}$ (When no history, search on the gradient direction)
for $i = |\mathcal{H}| - 1$ **downto** 0 **do**
    $\alpha_i \leftarrow \dfrac{\mathbf{s}_i^\top \mathbf{q}}{\mathbf{y}_i^\top \mathbf{s}_i}$
    $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$
end for
$\mathbf{r} \leftarrow H_0 \mathbf{q}$ ▷ Initial Hessian approximation
for $i = 0$ **to** $|\mathcal{H}| - 1$ **do**
    $\beta \leftarrow \dfrac{\mathbf{y}_i^\top \mathbf{r}}{\mathbf{y}_i^\top \mathbf{s}_i}$
    $\mathbf{r} \leftarrow \mathbf{r} + (\alpha_i - \beta)\mathbf{s}_i$
end for
return $\mathbf{p} = -\mathbf{r}$ (Search direction)

From Algorithm 3.2 we want to show the importance of a well estimated Hessian. The quality of this approximation depends directly on the amount of past iteration data stored, specifically the coordinate differences and gradient differences. If no history is used ($m = 0$), the method defaults to standard gradient descent, moving in the steepest descent direction at every step. In this case, the algorithm has no information about the curvature of the potential energy surface (PES), leading to potentially slow convergence, especially in regions with narrow or curved valleys. However, as more history is incorporated ($m > 0$), the algorithm gradually builds an approximation of the second-order curvature, allowing it to navigate the PES more efficiently. A well-estimated Hessian helps accelerate convergence by adapting to the local geometry of the optimization landscape. Nonetheless, increasing the history size indefinitely is not necessarily beneficial. While a larger history captures more of the past dynamics, it may also introduce outdated curvature information that is no longer relevant to the current region of the PES. This can lead to inefficient updates, making the algorithm less responsive to local optimization landscape. Therefore, an optimal history size balances the need for curvature information without including in the optimization process obsolete data. Another critical aspect to consider is the risk of premature divergence. If the initial geometry is far from a physically meaningful configuration, the gradients may be large and uninformative, causing the Hessian approximation to be inaccurate. In such cases, the quasi-Newton updates may lead to wrong search directions, making the optimization unstable.

The Wolfe conditions are checking that the new function value exhibits a significant reduction relative to the current value ($f(x_k)$) and that this increase does remains meaningful and that the algorithm does not take a step that could overshoot the minimizer. Those

conditions are respectively the *Arminjo Sufficient Decrease Curvature Condition*. Below is reported the specific implementation of Line Searches showed in the reference optimization book [16]. This implementation showed to have several serial computations presents like from line 10 to line 27 of Algorithm 3.3 or from line 8 to line 20 of Algorithm 3.4 (the two line searches method) are carried by all threads since are just scalar computations for each ligand and parallelization is not feasible in this computations. In contrast any vector operation and energy calculation is carried out in parallel. Even though we expect the main computational load to be in energy evaluation, as already discussed previously, approaching the porting of the solely energy evaluation for each ligand would require enormous overhead of launching several kernels (for every iteration of every ligand of the bucket, potentially many times). This motivates the low amount of threads per block (32). We mainly want to use warp primitives for any vector operation in the algorithm and energy evaluation, since the algorithm has the need of be synchronized, we can use $__syncwarp()$ CUDA primitives, without padding the atom's coordinate to multiple of 32.

Algorithm 3.3 More-Thuente Line Search

Require: Initial step size α, function f, gradient $\mathbf{g}$, search direction $\mathbf{p}$, parameters μ (sufficient decrease), η (curvature)

Ensure: Step size α satisfying Wolfe conditions

$\alpha_{\text{init}} \leftarrow \alpha$

$\Phi_0 \leftarrow \{0, f(\mathbf{x}_k), \mathbf{g}_k^\top \mathbf{p}_k\}$ ▷ Initial point

$\Phi_l \leftarrow \Phi_0$ ▷ Lower bound

$\Phi_u \leftarrow \{\alpha_{\text{init}}, f(\mathbf{x}_k + \alpha_{\text{init}}\mathbf{p}_k), \nabla f(\mathbf{x}_k + \alpha_{\text{init}}\mathbf{p}_k)^\top \mathbf{p}_k\}$ ▷ Upper bound

$\Phi_j \leftarrow \Phi_u$ ▷ Current point

Initialize: bracketed $\leftarrow$ false, stage1 $\leftarrow$ true

for $k = 1$ **to** max_iterations **do**

  Compute $\mathbf{g}_j = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ ▷ Gradient at trial point

  $\Phi_j \leftarrow \{\alpha, f(\mathbf{x}_k + \alpha\mathbf{p}_k), \mathbf{g}_j^\top \mathbf{p}_k\}$ ▷ Update current point

  if Φ_j satisfies Wolfe conditions **then**

    return $\alpha = \Phi_j.a$

  end if

  if Φ_j is not finite **then**

    break

  end if

  $\psi(\Phi) \leftarrow \{\Phi.a, \Phi.f - \Phi_0.f - \mu\Phi_0.d\Phi.a, \Phi.d - \mu\Phi_0.d\}$ ▷ Modified function

  if stage1 **and** $\psi(\Phi_j).f > 0$ **or** $\Phi_j.d < \min(\mu, \eta)\Phi_0.d$ **then**

    $\alpha \leftarrow \text{TrialValue}(\psi(\Phi_l), \psi(\Phi_j), \psi(\Phi_u), \text{bracketed})$

  else

    $\alpha \leftarrow \text{TrialValue}(\Phi_l, \Phi_j, \Phi_u, \text{bracketed})$

  end if

  bracketed $\leftarrow$ bracketed **or** (case $\in \{1, 2\}$)

  Update Φ_l, Φ_u, Φ_j based on α

  if bracketed **then**

    Safeguard α to ensure sufficient decrease in interval size

  end if

end for

return $\alpha = \alpha_{\text{init}}$ ▷ Fallback to initial step size

Algorithm 3.4 Backtracking Line Search

Require: Initial step size α, function f, gradient $\mathbf{g}$, search direction $\mathbf{p}$, parameters μ (sufficient decrease), η (curvature)

Ensure: Step size α satisfying Wolfe conditions

$\alpha_{\text{init}} \leftarrow \alpha$
$\Phi_0 \leftarrow \{0, f(\mathbf{x}_k), \mathbf{g}_k^\top \mathbf{p}_k\}$ ▷ Initial point
$\Phi_l \leftarrow \Phi_0$ ▷ Lower bound
$\Phi_u \leftarrow \{\alpha_{\text{init}}, f(\mathbf{x}_k + \alpha_{\text{init}}\mathbf{p}_k), \nabla f(\mathbf{x}_k + \alpha_{\text{init}}\mathbf{p}_k)^\top \mathbf{p}_k\}$ ▷ Upper bound
for $k = 1$ **to** max_iterations **do**
    Compute $\mathbf{g}_u = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ ▷ Gradient at trial point
    $\Phi_u \leftarrow \{\alpha, f(\mathbf{x}_k + \alpha\mathbf{p}_k), \mathbf{g}_u^\top \mathbf{p}_k\}$ ▷ Update upper bound
    if Φ_u satisfies Wolfe conditions **then**
        return $\alpha = \Phi_u.a$
    end if
    if Φ_u does not satisfy sufficient decrease **or** $\Phi_u.f \geq \Phi_l.f$ **then**
        break ▷ Enter zoom phase
    end if
    if $\Phi_u.d \geq 0$ **then**
        Swap Φ_l and Φ_u ▷ Enter zoom phase with inverted bounds
        break
    end if
    $\alpha \leftarrow 0.5 \times (1 - \alpha)$ ▷ Reduce step size
    $\Phi_l \leftarrow \Phi_u$
    $\Phi_u \leftarrow \{\alpha, f(\mathbf{x}_k + \alpha\mathbf{p}_k), \nabla f(\mathbf{x}_k + \alpha\mathbf{p}_k)^\top \mathbf{p}_k\}$
end for
Zoom Phase:
while $k \leq$ max_iterations **do**
    $\alpha \leftarrow (\Phi_l.a + \Phi_u.a)/2$
    Compute $\mathbf{g}_j = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ ▷ Gradient at trial point
    $\Phi_j \leftarrow \{\alpha, f(\mathbf{x}_k + \alpha\mathbf{p}_k), \mathbf{g}_j^\top \mathbf{p}_k\}$ ▷ Update current point
    if Φ_j satisfies Wolfe conditions **then**
        return $\alpha = \Phi_j.a$
    end if
    if Φ_j does not satisfy sufficient decrease **or** $\Phi_j.f \geq \Phi_l.f$ **then**
        $\Phi_u \leftarrow \Phi_j$
    else
        if $\Phi_j.d \times (\Phi_u.a - \Phi_l.a) \geq 0$ **then**
            $\Phi_u \leftarrow \Phi_l$
        end if
        $\Phi_l \leftarrow \Phi_j$
    end if
end while
return $\alpha = \alpha_{\text{init}}$ ▷ Fallback to initial step size

3.3. Minimizing work imbalance of bucket

The more the bucket gets bigger (the bigger the GPU), the higher the CPU computation for each bucket is. The solution proposed to parallelize the data preparation of buckets is to create **a stream per thread of the minimizer stage**. In this way, CPU preparation of buckets is executed in parallel between buckets, and as soon as one buckets frees the device, the other has already copied its data on GPU, and so, is ready for minimizing its bucket while the previous one copies the minimized coordinates back to host.

Since in kernel multiple molecules are minimized, we expect that if a molecule that is taking lots of iterations to convergence is in the same bucket of all remaining molecules that converged in few steps, the entire bucket is slowed by one ligand. Of course, the provided example is an extreme scenario, but more moderate situations may still occurs. From Nsyght compute we observed that *work imbalance* is of 16% as average in all blocks of the same subset dataset created previously.

In this section we explain the implementation of the bucketizer, that is responsible of how are molecules organized before feeding them into GPU.

We wanted to keep this stage the most light as possible, so that bottleneck of the pipeline should remain the minimizer. The aim of this stage has been to reorganize the *current* bucket before pushing it, in an online manner, without waiting for all ligands initial guesses before doing the minimization.

While we stressed out that the computational load's main responsible in ligand minimization is ligand size, it is not the solely responsible one. Energy evaluations of big ligands is computationally heavier than evaluating smaller ones, however, there is no information about the complexity of the objective function, that is the PES. Is a reasonable assumption that the needed time to minimize a molecule is a combination of the two factors.

A literature research about the problem of predicting the complexity of a small molecule using ligand's feature was conducted, with the remark that the both the model used to predict the complexity of PES landscape, and the feature extraction from molecule must be computational efficient. First, since our problem is to find the minimum on the PES, the complexity of the landscape is strictly correlated with the **conformational entropy**; it indicates how many low-energy conformation of a molecule are possible. Computing entropy is an extremely computational demanding challenge.

In "Understanding Conformational Entropy in Small Molecules"[4] by L. Chan et. al.

however, they managed to predict conformational entropy with a linear model, fitting the value against quantum chemical calculation's software results, with an $R^2 = 0.724$.

Below is reported the final model they used. Molecular features used are both standard features and prior-knowledge based one. Chemical relevance and assumption of the problem is not a scope of this work, therefore motivations are linked to original paper. N_{rotor} indicates the number of rotable bonds of the molecule. N_{Methyl} indicates the number of CH_3 groups, R_f^{Total} is the *ring flexibility*, and is a specific score classifying rings ability to adopt various changes, appositely created by authors. N_{SG} is the number of *specific groups*, again, a score created in reference paper. F_{Hbond} is a score for the flexibility of hydrogen bonds, and $F_{\pi-\pi}$ is the refers to non-covalent interactions between aromatic rings. These interactions occur due to the overlap of π-electron clouds in aromatic systems. Artificially created features showed to be the most complex to implement, in particular,

Table 3.1: Reported linear model obtained in reference paper "Understanding Conformational Entropy in Small Molecules" [4] with the ligand's feature in first column.

	Estimate	**std. Error**	**t**	**Pr(>|t|)**
Intercept	7.236	0.082	87.952	0.000
$\log(N_{\text{rotor}} + 1)$	9.932	0.052	190.792	0.000
$\log(N_{\text{Methyl}} + 1)$	13.024	0.034	378.978	0.000
$\log(R_f^{\text{Total}} + 1)$	-0.116	0.031	-3.784	0.000
$\log(N_{\text{SG}} + 1)$	-2.124	0.053	-39.849	0.000
$\log(F_{\text{HBond}} + 1)$	-3.082	0.038	-80.263	0.000
$\log(F_{\pi-\pi} + 1)$	-0.503	0.037	-13.719	0.000
R^2		0.724		

the *ring flexibility* and π-π *staking*. A complete explanation of how those features are retrieved is present in the paper's appendix about, with the SMARTS (SMILES arbitrary target specification) patterns too.

Since the model is fitted against conformational entropy of quantum chemistry software, and being the conformational entropy value sensitive to the force fields used, we rather use features directly to predict the iterations to reach convergence instead of using the paper coefficient of the linear model. We use the same subset of EXSCALATE dataset discussed in previous experiments, and check the correlation between iterations and feature values.

Figure 3.9 shows that *ring flexibility* is almost uncorrelated to iteration number. However, they are correlated as well in our case, most likely because of the difference force field used against the paper. All other features have some degree of correlation, therefore, we keep only the most correlated one with the iteration number. Considering many features in clustering algorithm induce curse of dimensionality, and considering correlated features

introduce bias on the distance metric. When two features are highly correlated, they essentially double count the same information. We highlight that the aim of the paper and ours is slightly different; we don't want to predict accurately the conformational entropy, but rather get an insight about the inner complexity of the objective function of the algorithm, in order to heuristically clusterize molecules with similar complexity. For this reason, we kept only rotable bond feature for bucketizing-clustering strategy to account the chemical complexity, while for the computational one we keep the *number of atoms*.

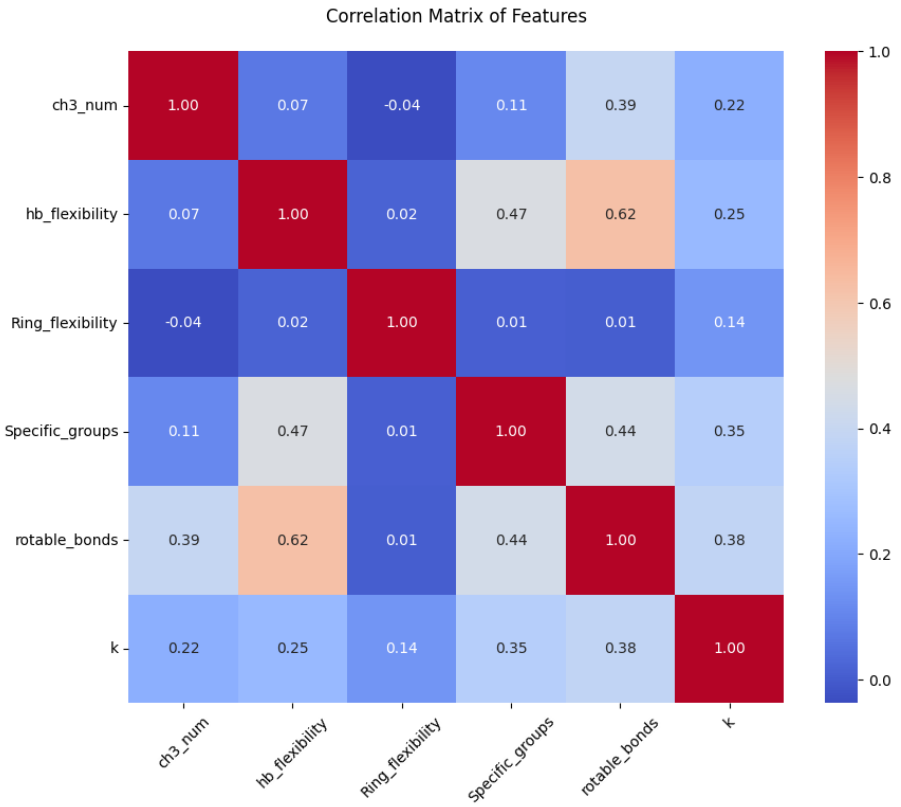


Figure 3.2: Correlation matrix of k (iteration number to reach convergence) and ligand features.

We designed a lightweight clustering strategy to efficiently organize molecules before pushing to minimizer stage.

Since the bucketizer operates in an online manner, processing molecules as they arrive without waiting for a complete dataset, traditional clustering algorithms that require a global dataset (such as k-means or hierarchical methods) are unsuitable. Instead, we use an incremental clustering strategy, where incoming molecules are dynamically assigned to existing clusters based on their feature similarity.

Before diving into the details of the clustering strategy, it is important to stress that the order of ligands in a single bucket is critical because of the GPU's wave-based execution. The kernel processes ligands of a bucket in waves (equal to the batch multiplier hyperparameter), each wave consisting in a number l of ligands, as determined from the out-of-kernel optimization paper [1]. This fixed grouping is essential: if even one ligand in a wave (or cluster) takes significantly longer to converge than the others, it can delay the entire kernel. Our clustering strategy, therefore, is designed to reorganize ligands so that those with similar computational loads are processed together, in the same wave, minimizing potential delays.

The implemented bucketizer is a lightweight, online clustering algorithm that organizes molecules into fixed-dimension clusters, each corresponding to the number l of ligands per GPU wave. Each molecule is characterized by a feature vector that, for reasons explained previously, is $\in R^2$ composed by the number of rotable bonds and the number of atoms.

When a new molecule is received, its feature vector is computed. If the bucket is empty, meaning that no clusters have been populated yet, the feature of the first molecule is used to initialize the center of every cluster. As next molecules arrive, the algorithm computes the euclidean distance between the new molecule's feature vector and every current cluster center.

The molecule is assigned to the cluster with the closest center. If the chosen cluster is not composed of l molecules, is added to that cluster. After the assignment, the cluster center is updated. This is done by adjusting the center in this way:

$$new_center = old_center + \frac{new_point - old_center}{n+1} \tag{3.1}$$

where n represents the number of ligands present in the cluster at that point. This approach ensures that the cluster center converges to the average features of its elements.

In situations where the closest cluster is already full, the algorithm searches for the next closest cluster that still has capacity. Once all clusters are full, the bucket is ready to be forwarded to minimizer stage. This algorithm minimizes the risk that a single, more complex molecule will bottleneck an entire GPU wave.

3.4. Design kernel structure

Remarking that our optimization is for the problem of generating several 3-D conformations, the mapping should be coherent to our expected results. In contrast to all existing

minimization softwares of the state of the art, that aim to be efficient on the minimization of single molecules or protein, we aim to reach an high throughput of big libraries of molecules used in VS. Optimizing single, small molecules on GPU is not the aim of this work. Instead, we look at the problem from another perspective; since GPU offers multiple level of parallelization, we want to exploit coarser parallelization among molecules, and finer parallelization for the single molecule. Hence, the chosen mapping is a **block for each molecule**; with blocks composed of 32 threads (a warp). As long as the computation was carried out on CPU, the bucket size is not influent on performance, we're computing buckets in parallel (and not molecules in bucket in parallel) so we don't want to keep a too high bucket size.

When porting the energy minimization on GPU following the mapping molecule-to-block, the bucket size becomes important. We want a bucket with amount of ligands that uses most GPU resources for the longest time; for this reason, we must calculate how many ligands we can process on GPU in parallel (single wave). In literature[1] they referred to this number as l, and they used largest multiple of this number that fits into GPU's memory. We set $t = 32$ the number of threads for a ligand. At this point, the number of active CUDA block that can be issued depends on kernel requirements, that are registers needed and shared memory. Using $cudaOccupancyMaxActiveBlocksPerMultiprocessor$ we retrieve how many blocks b can run on same SM. Therefore, the bucket size l is computed as:

$$l = b \times SM \frac{t}{ws} \tag{3.2}$$

where ws is warp size. In reference paper, this number is multiplied by a scalar, that must be ideally fine tuned, however, they showed no significant changes for the multiplier. Higher multipliers imply higher tailing effect for the last bucket (typically the ligands amount in input are not a multiple of the bucket size) of the dataset, allocating high number of blocks but calculating few ligands in that last kernel. For this reason, we tested multipliers from 3 to 8, without noticing significant differences (for each multiplier 5 tests with subset of EXSCALATE dataset, times difference was not statistically significant at any level by ANOVA). Therefore, we've fixed $batch_size = 6$.

Once we've defined the amount of ligands processed together in the kernel, we still must try to manage a bucket-parallelization, that was easily achieved in CPU by increasing the number of threads executing the batch minimization. While we have single GPU machines, we still have to perform data preparation prior to kernel execution; it consists in flattening arrays of coordinate and interaction.

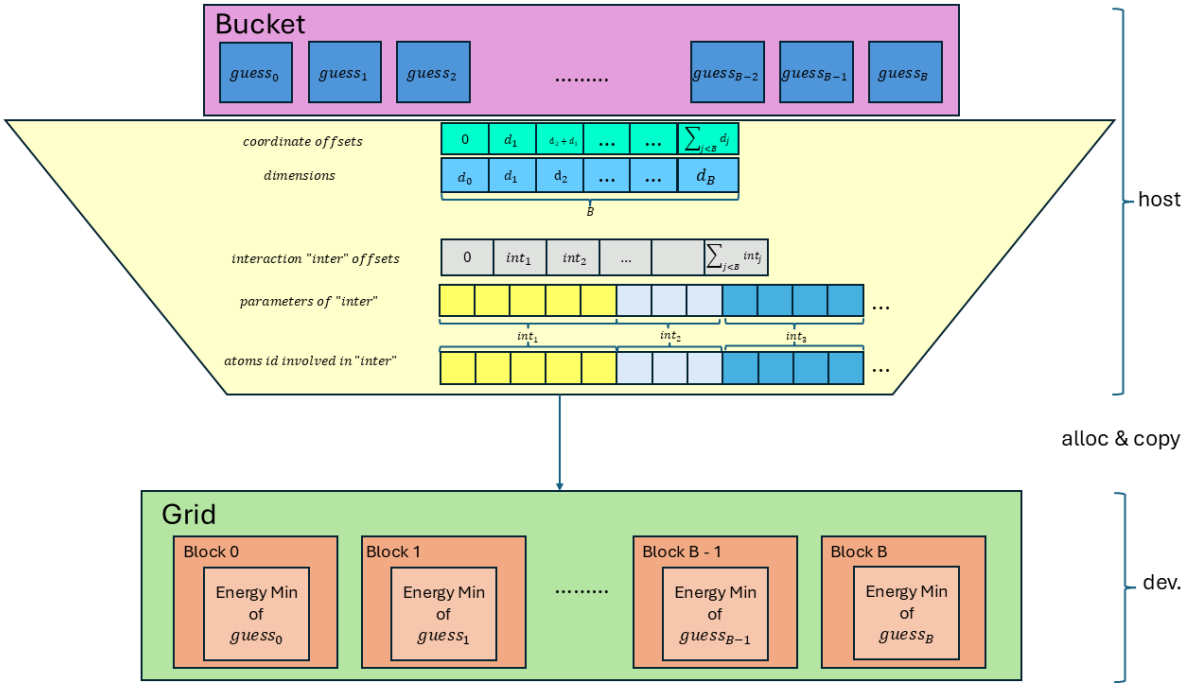


Figure 3.3: GPU kernel map of computation. One block minimize a ligand of the bucket. One general interaction "inter" is represented. In implementation, there are several arrays like this, once per interaction.

3.5. Kernel Optimization

Since we've used a local machine with a gaming GPU, that is equipped with float hardware arithmetic units (more detail in Section 4.1.1), we tried to reduce the double precision operations. Using only single precision is not appropriate for numeric reasons; for instance, exit conditions that checks gradient norms and residuals between iterations, have very low thresholds, changing them to floating point thresholds would not guarantee same convergence of the algorithm. In contrast, regarding coordinates, much higher thresholds are used, therefore, single precision approaches was investigated for the coordinate vector. The implementation consisted in using double precision for energy evaluations part since high accuracy is needed to converge, while for coordinate updates we've used single precision. For HPC-grade machine we've used double precision for every term. More discussions about numerical precision are available in results.

All dots product, norms are parallelized with warp-reduce primitives, exploiting high bandwidth of registers: each thread calculate its contribute and then a *reduction* is performed.

For the update of the hessian, each thread updates the index of the hessian array based

on its thread id. Same parallel pattern is also implemented for the direction vector.

Above mentioned sub-functionalities of the algorithm resulted in easy conceptual mapping to GPU, and by using *nsight compute* [5] no significant bottlenecks was found on any of these, in particular regarding memory accesses, being coalesced.

Main conceptual challenges for an efficient GPU kernel came up with non-bonded energy evaluation.

The first naive approach consisted in mapping an *interaction to a thread*. For this reason we first define what an *interaction* means in our context, and specifically, in our implementation. In 1.3.2 we defined energy contributions present in the MMFF94 force field. For each of those contributions, we define an interaction of that type. Each interaction involves at least two atoms, and no more than four. Also, each interaction has a set of parameters needed to describe the functional of the energy in relation with distances and angles.

Table 3.2: Table defining interactions, atoms involved, and number of parameters.

Interaction Name	**Atoms Involved**	**# of Parameters**
Bond Stretch	2	2
Angle Bend	3	3
Stretch Bend	3	5
Out Of Plane	4	1
Torsional	4	3
Electrostatic	2	3
Van der Waals	2	2

In light of this table, we can expand the mapping shown in Figure 3.5. For each interaction, "number-of-atom-involved" arrays are created (i.e. 3 arrays for *stretch-bend* interactions), one interaction offset for each interaction type (so that each block accesses to the interactions of its assigned ligand) and finally, "number-of-parameters" arrays are created. Mapping one interaction per thread, at this point, consists in threads calculating the energy contribution and gradient of pairs, triplets and quadruplets of atoms.

When a generic interaction must be computed, coordinates of the $2, 3, 4$ atoms are needed to compute distances and angle with respect to their respective 3 coordinates. We can look at the energy evaluation as a reduction pattern; threads of the block scan to the interactions array (of every interaction, all threads operating wisely on each interaction together), evaluate their intermediate forces, and then reduce after having calculated all interactions of all type.

The derivative is computed to update the gradient, and must be evaluated with respect to the x, y, z coordinates; those are then written in the gradient vector. For an interaction involving atoms at index a and b, the gradient is updated at position $3a$, $3a+1$, $3a+2$ and $3b$, $3b+1$, $3b+2$, respectively for coordinate x, y and z of the two atoms, same concept for more atoms.

3.5.1. Non-bonded interaction problem

The building of interactions array is carried out on CPU, because getting the force field parameters consists in consulting *ligen-dict*, that consists in lots of hash-tables with keys as atom type and parameters as value. Implementing this part on GPU would mean create scattered accesses and branch divergence because of chains of *if* statements that checks atom typing to further fine-tuning parameters (for instance, rings atoms are treated in different ways than plain atoms, or distinctions are made for *donors* and *acceptors*).

However, the way in which those interactions are built affect GPUs performances. The reason relies in the fact that when we access to the array shown in Figure 3.5 called *atoms id involved in "inter"*, we then access both the position vector (to get the coordinates needed to compute the energy) and the gradient vector **at those indexes**. Therefore, if indexes in that arrays drive to uncoalesced accesses, performance of GPU are affected.

We now briefly describe how interactions arrays are built. Bond stretch interactions array are built by iterating over edges of the graph, so the order of adding depends on the order induced of edge list of *boost*[3]. All remaining bonded arrays interactions (angle bend, stretch bend, torsional and out of plane) are built by iterating over vertices, checking their out-degree (if correspond to at least 2 for angles interaction and 3 for torsional and out of plane), and finally add atoms and parameters involved in interaction. Electrostatic and Van der Waals interaction followed same pattern; figuring out a full n x n interaction matrix, the order of interaction is a *row major*, where rows are source atoms and columns are target. A non bonded interaction is considered only when atoms are separated between at least 3 bonds. This is not only a computational trick to scale down the interaction number, but also necessary since MMFF94's parameters for Van der Waals are fine tuned for this scaling (called *1-4 interaction scaling*). They are not currently differentially scaled in MMFF94. Extremely close atoms interactions are ignored to prevent the distance from approaching zero, as both Van der Waals and Electrostatic potential tends to infinity at very short distances. Since electrostatic and Van der Waals followed the same order of processing, considering the same atoms in their interactions, we merged those two and called the energy contribution *VWE (Van der Waals Energy)*. For non charged atoms the

charges are just set to 0, nullifying the electrostatic force and gradient, but considering Van der Waals ones.

All the other bonded interactions scale linearly with the number of atoms ($O(n)$), while non-bonded interactions scales quadratically. For each atom, we should (theoretically) consider the interaction with all other atoms. We discussed about the bottleneck of the minimization stage being the energy evaluation, and being the non bonded interactions $O(n^2)$ in complexity, hence, the main interest was invested in optimizing *non-bonded interactions*, that are Van der Waals and Electrostatic.

An immediate trivial solution could be to scale down the number of interactions. At each step, depending on the conformation of the molecule reached at that iteration, we may not need to calculate all pairs of atoms. Imagining that a molecule reach a conformation where is very elongated, from end-to-end of the molecule may exists a near-to-zero contribution of non bonded energy.

This is a well studied approach in computational chemistry, and there exists several ways to reduce the number of interactions at each energy evaluation. All those methods are based on the fact that, Van der Waals potential falls off very rapidly with distance. The most popular way to deal with the non-bonded interactions is to use a *non-bonded cutoff*. When a cutoff is employed, the interactions between all pairs of atoms that are further apart than the cutoff value are set to zero. By itself, the use of a cutoff may not dramatically reduce the time taken to compute the number of non-bonded interactions. This is because one would still have to calculate the distance between every pair of atoms in the system simply to decide whether they are dose enough to calculate their interaction energy. Depending on the minimization algorithm, developers should observe over how many iterations the neighbors list doesn't change, and based on that, compute the energy only on that sublist of atoms. This approach, however, heavily relies on the minimization algorithm used (different algorithms updates more heavily the atoms coordinates than others) and on the type of system minimized. Moreover, keeping a dynamically sized atom list for each atom is not at first glimpse a GPU-friendly approach.

GPU's is notoriously limited by bandwidth. The implementation proposed in this thesis, indeed, aims to be efficient on GPU hardware, taking care of memory accesses. The naive starting approach consisted in assigning each interaction to a thread, using atomic operations on gradient vector updates, since interactions in a general order of processing could share either source or target atom of the interaction. If two interactions have same source or target atom, two threads would collide to update the the same position of the gradient. If the interaction matrix is arranged in *row-major*, considering interactions

only with target atom id strictly greater than source atom id (to not consider duplicates, basically storing only the right upper diagonal part of the matrix), each thread would need the same source atom coordinate of other threads to compute the energy of that interaction, and would update the gradient in the index corresponding to the source atom too, resulting inevitable atomic operation on the source atom's index of the gradient.

We'd also do $3n(n-1)$ read accesses to the position vector (three for each coordinate, and n being the number of atoms in the molecule), that is stored on global memory, and $3n(n-1)$ read-write atomic operations to the gradient vector. Last but not least, those accesses to position vector and gradient are inevitable non-coalesced: to give more context, an example is provided with real numbers. Figuring out a molecule with 100 atoms (hydrogen included), thread t may calculate interaction between atom with index 1 and 97 and thread $t+10$ interaction between atom with index 2 and index 7; to do this, thread t access position vector at indexes $3, 4, 5$ and $291, 292, 293$, while thread $t+10$ at $6, 7, 8$ and $21, 22, 23$. Those two threads are in same warp, but the caching of the accesses of atom's index 97 is unused, even if future (many) interactions at that index and above will be calculated.

An important remark to keep in mind from now on, is that this current implementation consider only **1-4 scaling interaction**. Consequently, each row of the matrix may potentially have different size. This create *impossible-to-predict* access patterns to memory. Excluding interactions is beneficial for computations, since the fewer the interactions, the fewer the energy evaluation, but for how the GPU is designed, it does not mean that we're exploiting the GPU architecture.

This was observed also from *nsight compute* profiling results of the first implemented version, where we considered the interaction evaluation as described above. In particular, target atom accesses exhibited a 12% inefficiency in global memory transactions due to non coalesced pattern, with gradient vector accesses showing same percentage. Regarding atomic operations on the gradient vector, they contributed to a 1.6% stall in warps.

3.5.2. Optimizing non-bonded interactions

We start by defining as n the number of atoms in a molecule, as $m^* = 32 * \lceil n/32 \rceil$ the padding of each row of the interaction matrix. We define also the atoms amount smallest submultiple of 32, that is $m = 32 * \lfloor n/32 \rfloor$ We also define a *dummy interaction* a non-bonded interaction not parametrized by MMFF94, that is for atoms separated by less than 3 atoms. For a dummy interaction, all parameters of Van der Waals and Electrostatic are set to 0. By looking at eq 1.9 and 1.10, we can see that if a parameter is set to 0, the

energy will be 0 too (as well as the gradient). A straight forward approach would be to we pad each row of the interaction matrix to m^*, obtaining a square matrix of this size, considering any interaction excluding only the diagonal (no interaction atom-atom since is trivial), and assign each row to a thread, writing contributes on gradient only for source atom (since by considering both (a, b) interactions and (b, a) if we write derivatives of both atom we'd considering twice). In this way, we don't need anymore atomic operations on gradient, and each thread would access only once to its source atoms and target atoms are accessed uniformly for all threads, in same order, hence, coalesced. This implementation, however, has two fallacies. First, for dummy interactions we'd need also dummy atoms. Since we don't want warp divergence, for dummy interactions we should create different dummy target atoms for each thread in the warp. At the same time, those dummy atoms must be real atoms, present in the molecule, since we must access their coordinates. This constraints those dummy atoms to be indexes of the vector and gradient array. At the same time, we should keep accesses coalesced; consequentially, we considered dummy atoms fixed as the last valid interaction's target atom. The second, more hard to avoid, caveat, has been the energy evaluation. We should consider the energy only for the source atom (once per interaction). But using the above explained procedure, we are calculating twice the force. Checking to accumulate energy value only when $id_{source} < id_{target}$ we are introducing *warp divergence*. For this reason, this approach was initially implemented but then discarded, as we didn't find a way to avoid warp divergence.

The final implementation exploits the fact that the force and energy of a pair are symmetric. Formally:

$$E_{nonbonded}(a, b) = E_{nonbonded}(b, a) \qquad \forall a, b \in \{atoms\} : a \neq b \tag{3.3}$$

$$\nabla E_{nonbonded}(a, b) = \nabla E_{nonbonded}(b, a) \qquad \forall a, b \in \{atoms\} : a \neq b \tag{3.4}$$

This property allows to flip the triangular matrix. Flipping the interaction matrix brings several advantages. First of all, all the threads in warp processes always both **different target and source atom** at the current iteration. This allow us to accumulate gradient's value to registers before writing to memory, without any need of atomic operations. A second perk is that source atom positions are uniquely handled by specific threads in the warp, and target atom's coordinates are shared among threads registers using *shfl* primitives without reading them from memory again.

Moreover, each thread exits the non-bonded energy evaluation at the same moment of

the others, without warp divergence.

The parallelization approach proposed aim to minimize the most as possible the number of uncoalesced accesses and atomic operations, since reducing them completely would mean to create dummy atoms to pad the atoms amount to a multiple of 32, with consequent dummy coordinates and dummy interaction parameters, eventually increasing too much the overhead. For this purpose, we do register pre-fetching of atom's coordinates only on the $[m, m]$ sub-matrix.

A first explorative pre-fetching stage is performed; the warp performs $\lfloor n/32 \rfloor$ steps to pre-fetch all the possible coordinates of the molecule. Each thread access unique source atoms and target atoms in the current interaction.

In figure is shown the pre-fetching stage of a molecule having $m = 64$; in this case we have 2 iterations for the warp to visit all atoms in the $m \times m$ submatrix. A warp iteration is identified by a main color, and a thread in that current warp iteration is identified by a color shade.

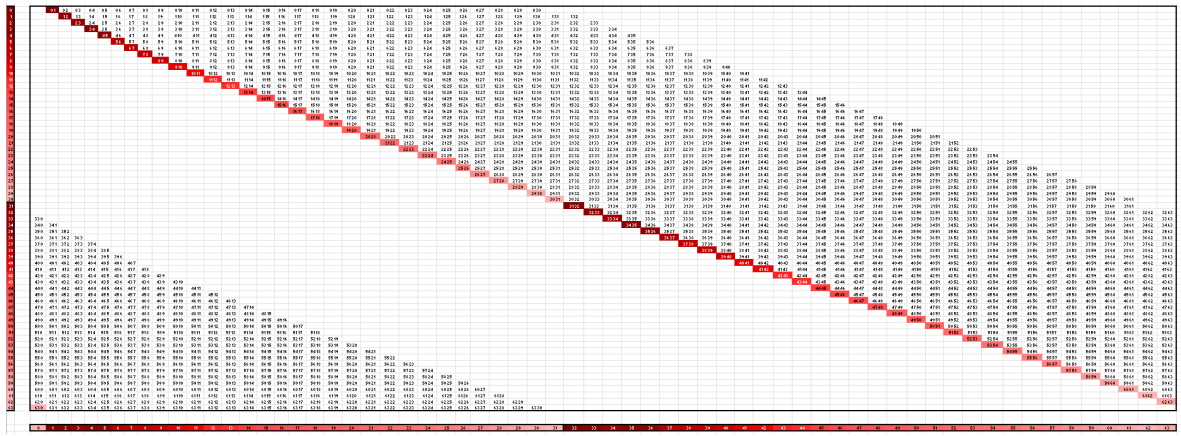


Figure 3.4: Prefetching atom coordinates, storing coordinate values in register. Source and target atom handled by thread are identified respectively by the left column vector and bottom row vector, with the responsible thread encoded by its color.

This pre-fetching allow to visit all atom's coordinate in $m/32$ steps per thread, accessing memory in a coalesced pattern. Since each thread now computes the gradient of unique source atoms (identified by their *thread id*), also the intermediate accumulated gradient value is kept in register. Regarding the target's atom gradient value, is also calculated and kept in appropriate registers. This is possible only because at each iteration, each thread accesses to *unique* sources and *unique* targets atom. For how interactions are reorganized, the uniqueness property of source atoms is valid for every iteration. At each iteration of

energy evaluation, indeed, we are able to assign to every threads a specific (equally sized for all threads in warp) subset of atoms, directly identified by their *thread id*.

After the pre-fetching, the computation of the remaining of interaction of the $[m, m]$ submatrix is carried out using shfl primitives: each thread t accesses to target's atom coordinate that thread $t+1$ used as target atom at previous iteration. At the same time, also the gradient evaluation of the target atom is shared among threads in the warp, so that the accumulation can be done in registers without writing $3m(m-1)/2$ times to memory, but only $3m$.

Table 3.3: Global memory access to gradient and position of the new strategy compared to the naive, row-major organization of interaction matrix. The without strategy accesses are approximated values since real values will be slightly lower, since are considered only atoms separated by at least three bonds. With strategy values are instead exact, because of dummy interactions.

Atom ranges	**Memory Access without strategy**	**Memory Access with strategy**	**Reduction factor**
[32-63]	1488	96	15x
[64-95]	6048	192	31x
[96-127]	13680	288	47x
[128-159]	24384	384	63x

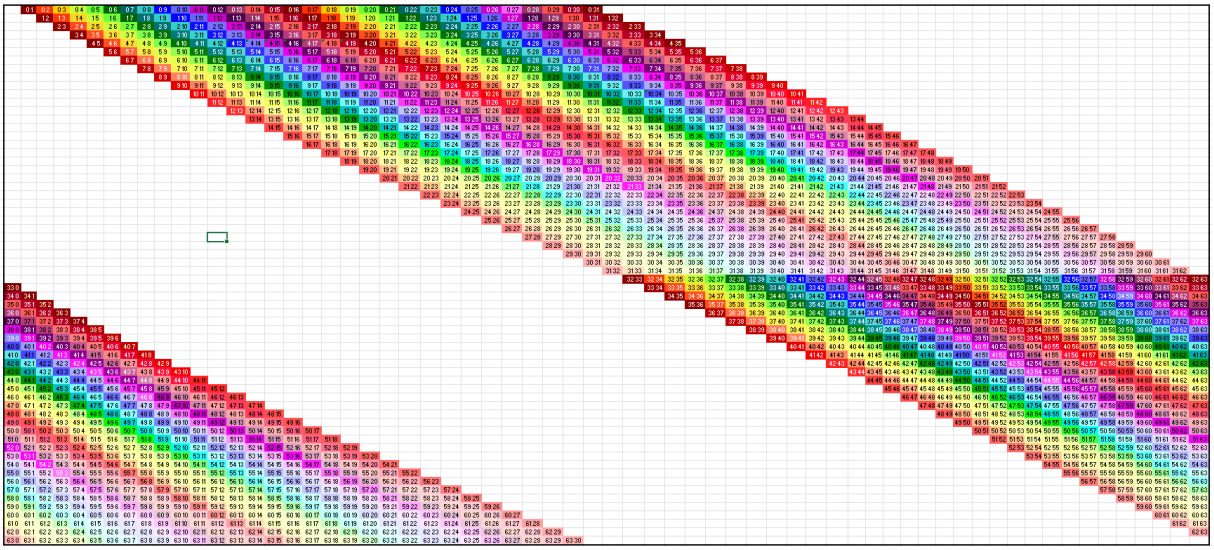


Figure 3.5: Reorganization of non bonded interaction matrix. Each warp is identified by the main color, than each thread inside the warp is identified by a its warp color, but with different nuance. With this picture we aim to show how target atom is shared among threads from previous iteration gradient evaluations.

In the below picture of a small example of a ligand with $m = 64$, is easier to see how sharing of atom's coordinate happens between iterations over the interaction array. Sharing among threads is indicated by green arrows, toward the target thread (thread index is the row of those columns, resetting after 32 since is the new iteration). is indicated by arrow For iterations up to the $m/32 - th$ coordinate values are prefetched into registers. Source atoms remain fixed while target atoms coordinate are shared to the next reponsible thread, together with the accumulated gradient value up to that iteration.

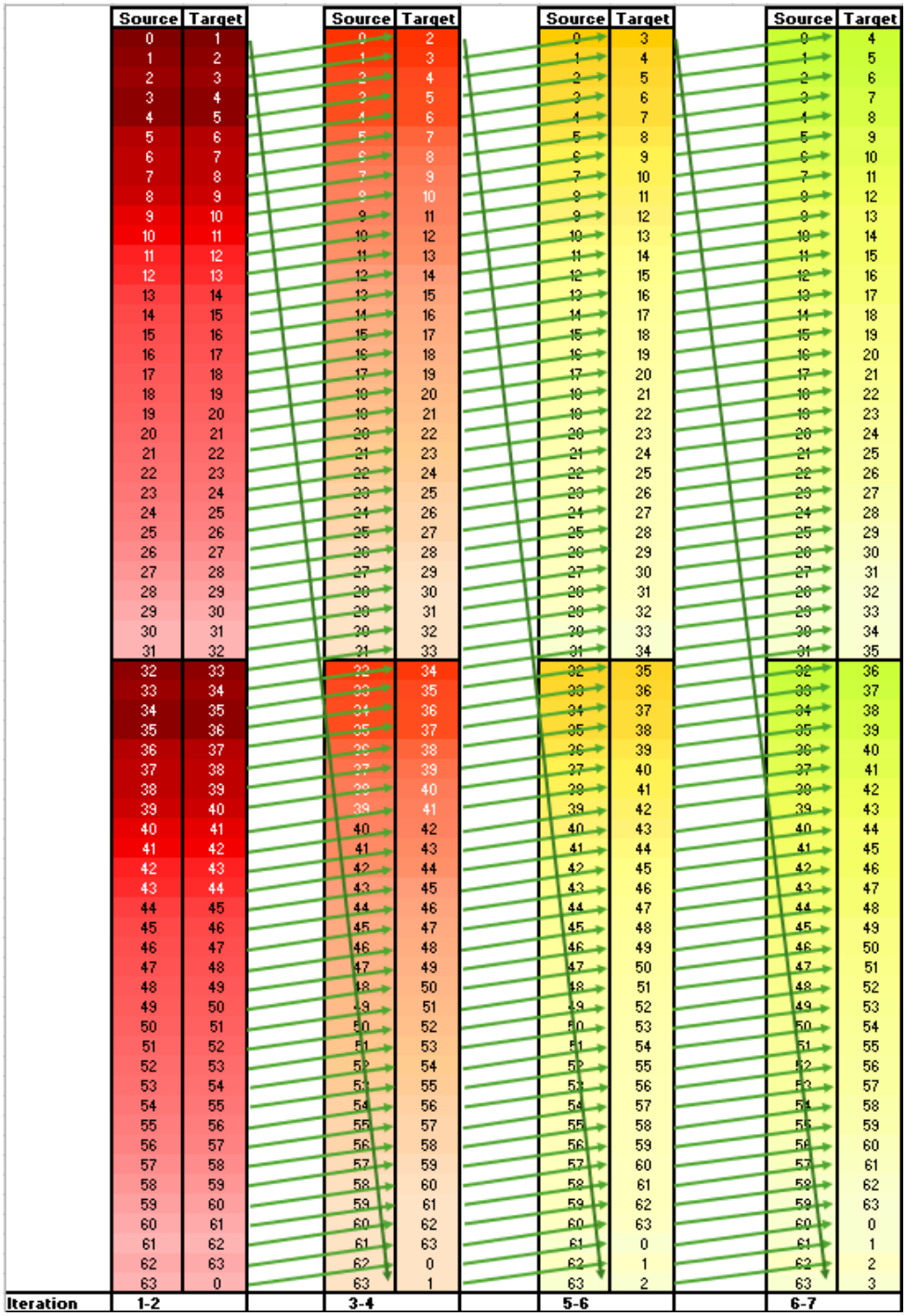


Figure 3.6: Warp iterations and sharing pattern after each interaction evaluation. Using the same color of the interaction matrix to show the order of processed interactions.

The final reduction is then performed on the index of the latest shared target atom indexes, with its respective computed-and-shared gradient value. Similar reduction is performed at source atom's index, but simply directly using thread index being source atom index constant for the thread.

We notice that up to this point, we've omitted the last diagonal, that visits atoms $[0, \frac{m}{2} - 1], [1, \frac{m}{2}], .., [\frac{m}{2}, m - 1]$. If we consider that using the same circular strategy, we'd write twice the same contribution to the same atom indexes. Is more clear from Figure 3.7. In that case, if we'd consider $[0, 32], [1, 33], .., [32, 63]$ and then circular indexes on left: $[32, 0], [33, 1], ..]$ we'd double count energy and gradient. For this reason, last lower column is not added, so that the final shuffle is not occurring, ensuring correctness of the evaluation step.

We still must handle remaining interactions; this strategy handled indeed the sub-matrix sized m. Those remaining interactions are handled with other 2 strategies. One strategy still uses registers in a simpler way than the above explained one, and the other strategy falls back to atomic operations and uncoalesced accesses (but since this strategy is acting on a very small number of interactions, we suppose performance side effects are negligible).

Say that Figure 3.7 represents the subset of the interaction matrix of a molecule with $n = 70$ atoms. Leftovers interactions consists of a $m \times (n - m)$ matrix plus a small *upper-triangular* $(n - m) \times (n - m)$ matrix. The small matrix is handled in the same way of the original naive implementation. We'll have some synchronization on writes on gradient indexes, but from Nsight Compute profiling few samplings was obtained so we expect not to be anymore a bottleneck.

The rectangular matrix instead, have another property that we can exploit to avoid intermediate writes for every thread, and use always a *reduce* pattern. Regarding rows writes, nothing changes conceptually: each thread is still responsible of writing on its assigned row index for the gradient. Regarding columns, in contrast, the same pattern previously used cannot be applied; column's index will overlap in any case within the gradient, since we don't have a large column amount but we're restricted to a small index set, that is $n - m$ possible choices for columns. A completely opposite approach in structure is used, at each iteration, all threads in the warp access to the same target's index coordinate. For rows, no access is done since result was already stores in registers from previous strategy.

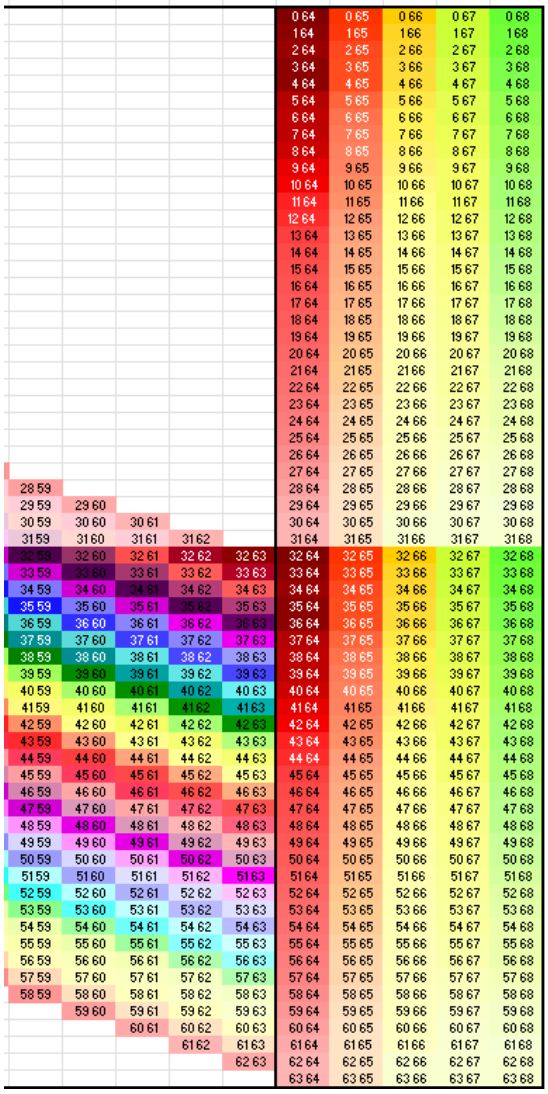


Figure 3.7: On the right we see an example of leftovers interactions handled using **column reduce pattern**. Same color encoding with shades used in previous figure.

Each thread at the end of one iteration shuffle its gradient accumulated value for that column to *thread leader* 0, that is responsible of the writing. Accesses of coordinate vector still exploit cache because each thread access in read only the same index of the array, and future iteration's access are cached for sure, since we're working with a small amount of columns and the transaction to make the warp caching is of 128 bytes, that is 16 double values. Implementing the column reduce only strategy for the whole matrix would not exploit the symmetry of the interaction matrix, having two threads performing the same operations in the warp.

3.6. Parameter selection and final computational considerations

While in CPU memory is almost unlimited in modern architectures, in GPU is not. Moreover, memory accesses are the slowest transfer in the GPUs. So, while larger m provides a better approximation but requires more memory, a smaller one reduces memory usage at the cost of less accurate approximation of the PES, considering it more rugged. Increasing m beyond a certain point yields diminishing returns in terms of convergence speed. Some attempts were made by tuning history depth depending on complexity of the molecule (in Section 3.4.2), but without any promising results. For this reason, any modification on m was not considered in the final implementation.

The choice of parameters, like tolerances of exit conditions, maximum iterations and history depth, is taken from LiGen-pre module and is not changed. The reason of this is that those parameters were fine-tuned on a dataset that, very likely, was much bigger than what i could have access, ensuring reaching convergence and good quality of coordinates on a vast library of ligands. Investigations on history depth was carried out, however, there isn't a clear correlation between history depth and iterations needed to convergence [2]. More in-depth analysis about numerical accuracies and speed of convergence fall outside the scope of the thesis, even though they are important in terms of reached throughput.

Using well tuned parameters of the LiGen application, we have that line searches (both More Thuente and Backtracking), potentially needs to evaluates up to 80 times the energy of the molecule, per iteration. This number is coming from the $k_{max} = 20$ of the line search More-Thuente and $k_{max} = 30$ of the backtracking (backtracking can, in general case, execute both the forward and backward steps). The exact number of energy evaluations depends on the convergence of the line search methods. As previously explained, we didn't want to change hyperparameter of the algorithm.

4 | Results

In this chapter a description of experimental results is provided. Not only final version's implementation results are shown, but also intermediate approaches that allowed to reach the final results, as well as approaches that aimed targeting different architecture with different strategies and pipeline configurations.

Performances of the final implementation depends on both processor and GPUs hardware used. Results of main interests are the ones obtained with final target machine, however, some intermediate results obtained using local machine are also provided for comparison purposes.

4.1. Experimental setup

This section reports information about the used hardware and the dataset that we use to measure the application performance.

4.1.1. Hardware description

The local machine is an ACER PEDATOR mounting Windows 11 WSL with Ubuntu 22.04.4 LTS, 16 GB GDDR6 RAM, CPU i9-13900HX with 24 core running at 2,20 GHz. The GPU is an NVIDIA GeForce RTX 4070 with 8GB.

One remote server, from now on refered to as *accento* is, as well, running Ubuntu 22.04.4 LTS. Equipped with 32 GB RAM, AMD EPYC 7302 with 20 core running at 2.99GHz. The GPU is an NVIDIA A100 SXM4 with 40GB. The target machine, from now on referred to as *LIGATE* has same characteristic of *accento* regarding GPU, but is equipped with 64 cores.

One key distinction between gaming and HPC-grade GPUs, such as the NVIDIA A100 SXM4, is that the latter ones are optimized for high-precision computations. These GPUs support double-precision floating point arithmetic (FP64) in hardware, making them well-suited for scientific computing that require high numerical accuracy. In con-

trast, consumer-grade GPUs are primarily optimized for gaming workloads that rely on single-precision (FP32) or mixed-precision (FP16) arithmetic. While they can still handle double-precision calculations, their FP64 performance is significantly lower due to software implementation.

4.1.2. Dataset

Results are obtained from the execution of the pipeline from the EXSCALATE [6] dataset (briefly introduced in Section 3.1 for the profiling of the application). Three datasets organized in ranges of molecular weights. Molecular weight was observed to be correlated with number of atoms, but also include the information about chemical properties of molecule, and so, also about ruggedness of the PES.

4.1.3. Profiled measures

We've profiled, in two different ways, the application every time an optimization was implemented. First way consists in analyzing the queue sizes of execution to identify bottleneck. For this option, we sampled randomly a subset of 20000 ligands from the dataset with heavier weight being the one most complex computationally. The second way regards GPU optimization, and consists in using Nsight Compute to retrieve metrics. Profiling the execution with full dataset would take too much time, since it requires the compilation in *Debug* mode and not in *Release*, hence without optimizations. For this reason, we extrapolated a subset of 20000 ligands from the dataset with MW $\in [330, 500]$.

For this purpose, LiGen has been profiled using *gprof* (a famous GNU profiler). It provides the **Percentage of Total Time**, that is how much of the total execution time of the program is spent on a function. It represents the cumulative time that the program spends executing that function, including the time spent on other functions it calls. The **Exclusive Time** instead is the time spent inside the function, excluding the time on its child functions.

The application is profiled using a small randomly generated subset from the EXSCALATE datasets [6]. Three datasets with ranges of molecular weights are available. The generated dataset consists in sampling 600 random SMILES from each of the following MW ranges $[1, 330), [330, 550), [550, \infty)$. The dataset used in profiling consists, finally, in 1800 SMILES. On the table below are reported the results of the profile, excluding secondary functions that took very low time percentages (% Total Time < 0.5).

Table 4.1: Performance profiling data for the serial original LiGen application.

Function Name	% of Total Time	Exclusive Time	# of Calls
get_mmff94_properties	85.20%	0.00%	2527
mmff94_atom_type	17.25%	0.00%	2151
LBFGS	**87.21%**	**2.19%**	10297
calc_energy_and_gradient	84.53%	0.02%	1239367
van_der_waals	**46.47%**	**46.47%**	1239367
torsional	12.68%	12.68%	1239367
stretch_bend	9.12%	9.12%	1239367
angle_bending	7.80%	7.80%	1239367
electrostatic	4.21%	4.21%	1239367
out_of_plane_bending	3.24%	3.24%	1239367
bond_stretching	0.97%	0.97%	1239367
get_coordinate_guess	10.54%	0.00%	2527
set_coordinates	10.34%	9.07%	2527

4.2. CPU parallelization results

In the first chapter we show results obtained by the pipelined approach. As explained in Section 3.2 it consisted in decomposing the conceptual stages in parallel since they are independent one from the other.

4.2.1. CPU Pipeline parallelization

The implemented pipeline's throughput depends on the slowest stage of the pipeline, since is the one that frees its queue more slowly than others, that ideally would go faster if the bottleneck stage wasn't present. From the shown profiling in Table 3.1, we could have imagined that the bottleneck stage was the minimizer, since is the stage performing the energy evaluation. This is proven also by analyzing the pipeline evolution; using the dataset discussed in Section 4.1.2 for this purpose. Figure 4.1 shows the execution of the pipeline with this dataset. Represents the amount of ligands on queues of each stage (identified by different colors). Every second, the queues sizes are logged and represented in the ordinate. The pipeline is configured with one threads per stage, $t_p, t_b, t_m = 1$. Since the information about queue's size is not influencing the total time, but only making the queues top capacity higher, small queues (200 ligands per stage) are used. Coherently with the results of the profiling, the minimizer is the bottleneck also in this parallel implementation, because of the immediate filling of its queue and the subsequent (orange

almost vertical line slightly later than the green one) filling of the bucketizer, that is before the minimizing stage. This is indeed what happens in first seconds of the picture: the minimizer can't stay at the pace of the guesser. This scenario is represented by the spike of the green line, so, the minimizer's queue can't be emptied at the same pace of other stages. It then follows the spike of the orange line, that is a back-pressure effect caused by the minimizer: the bucketizer would forward buckets at frequencies that the minimizer can't sustain, and for this reason, it start to accumulate buckets in its queue. The guesser is eager of work, continuously asking for molecules to provide initial guesses since its queue immediately fills up, and when it starts to become empty, leaves the bucketizer and minimizer the role of handling remaining molecules. The minimizer can't stay at the pace of the bucketizer too, as indicated by the immediate skyrocket of the green line in first seconds of execution. Therefore, the latter queue size (orange line) increases as long as it fills up, feeling the effect of back pressure caused by the minimizer. Back pressure is caused by the minimizer, since the bucketizer, in order to push buckets, should have its output queue free, but it is not, since the minimizer is still working on previous elements.

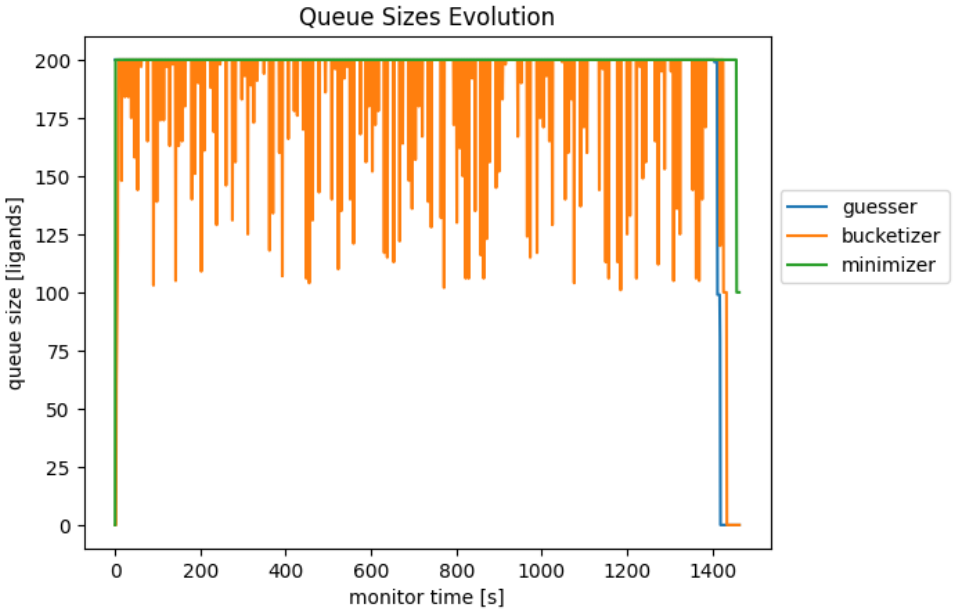


Figure 4.1: Pipeline's stages queue size in time. Queue size of each stage logged every second.

The strategy to face the bottleneck has been to iteratively increase the number of threads for the current bottleneck stage. First, threads of the bottleneck are increased, queues are analyzed to understand if there is a new stage as bottleneck. This behavior can happen due to the increase of threads. Anytime that a bottleneck is faced, this procedure is

repeated. This iterative approach was executed only for the subset of the heavier dataset for time constraints, but same configuration of threads was ensured to be conceptually valid for all the other datasets.

This iterative fine-tuning approach is shown with an heat map where each cell's value is the speedup obtained compared to the serial version. By following the colors to reach the darker one, we follow in the optimal configuration of thread allocation (higher speedup of the matrix). At the first row we see that even if we increase the number of guessers, the throughput is dictated by the minimizer stage and for this reason never changes in that configurations set. Increasing the number of minimizer's threads ends up reaching a plateau of speedup value: in the first column there is advantage up to the $t_{minimizer} = 16$, but then we have to shift toward configurations with higher number of guesser (toward right).

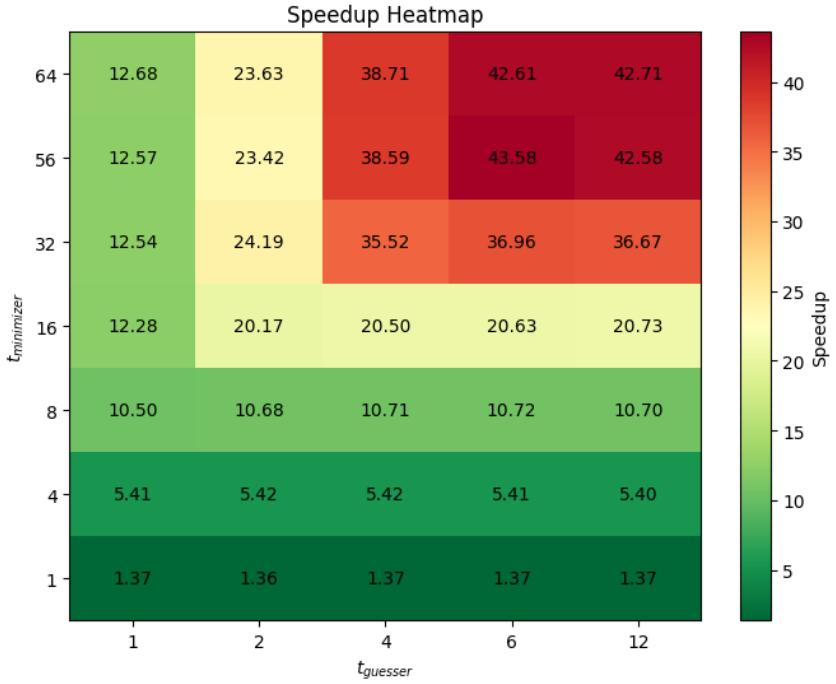


Figure 4.2: Speedup variation depending on pipeline configuration.

With the optimal configuration we have allocated all available threads of the LIGATE machine: 56 $(minimizer)$ $+ 6$ $guesser$ $+$ 1 $(reader) +$ 1 $(bucketizer) = 64$. Creating virtual threads introduce context switches between threads, slightly increasing time of execution: this happen in all cell's with sum of threads higher than 64 (top-right).

In the optimal configuration we were able to make the bottleneck a combination of the two stages: guesser and minimizer alternates in dictating the throughput of the application.

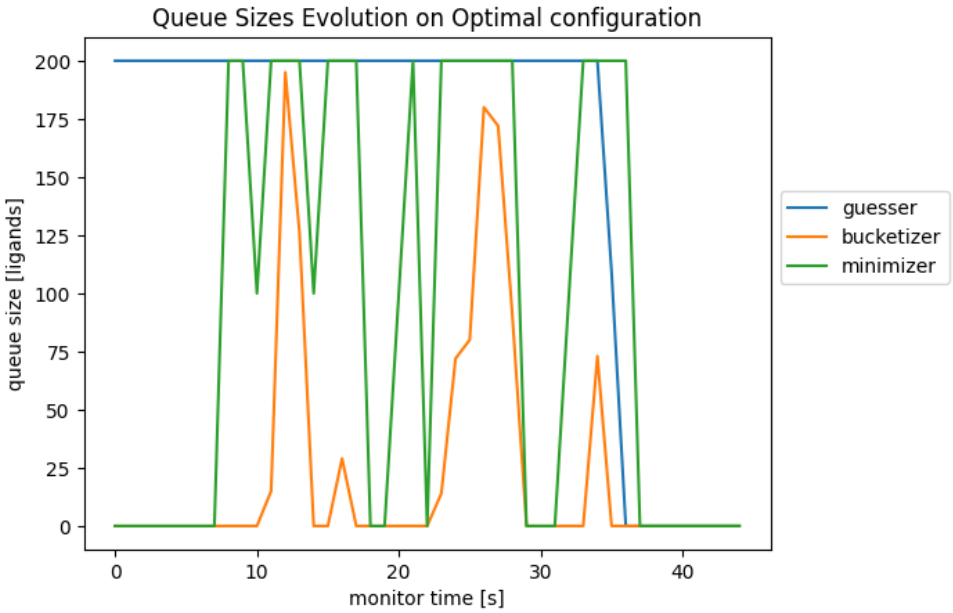


Figure 4.3: Optimal configuration queue evolution.

4.3. GPU porting Analysis

In this section we measure the both performances benefit of offloading the minimization step to GPU and analysis of the strategy, incorporating in the analysis profling results before and after any optimizations.

4.3.1. Mixed precision Effect

We investigate our mixed-precision approach of coordinate in single and energetic terms in double precision; whether it affect the quality of expanded molecules and its impact on performances. For this reason, we've evaluated the energy value of converged molecules and their 3-D obtained coordinates of both versions (using double precision for every term and the mixed-precision version). RMSD (Root Mean Square Deviation of atomic positions) is then calculated using MDAnalysis [14] between molecules obtained using the two methods. Hydrogen atoms are also considered.

RMSD represents the average deviation of atomic positions between the two molecular structures. A low RMSD indicates minimal structural differences, suggesting that the mixed-precision approach maintains molecular integrity. Conversely, a high RMSD would suggest that using lower precision for coordinates introduces significant distortions. From docking results interpretability studies [15], we have a general rule of thumb about RMSD values. For minimized molecular structures, $\mathbf{RMSD} < \mathbf{0.5}$ **Å** indicates nearly indistin-

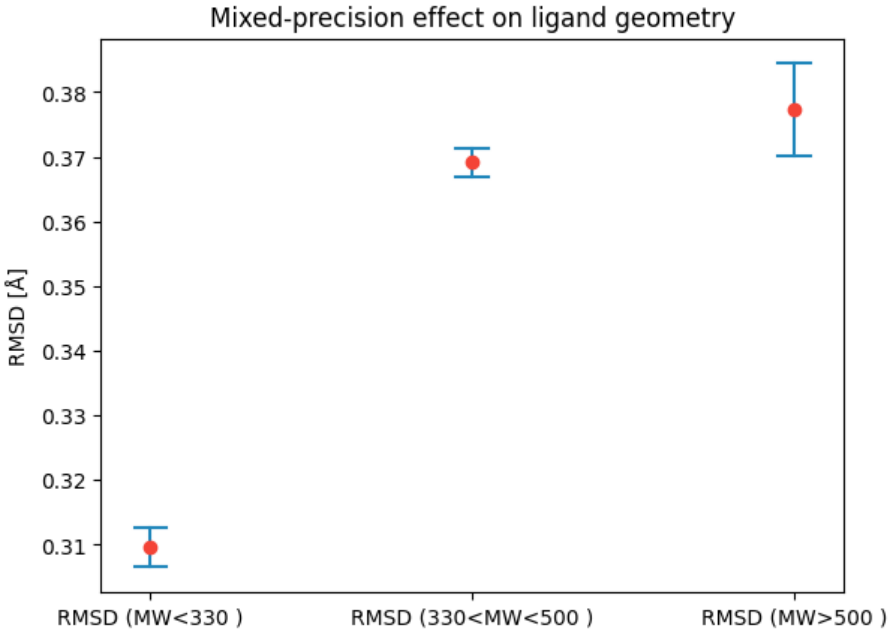


Figure 4.4: Confidence interval for RMSD between ligands minimized using float and double coordinates.

guishable geometries and almost no chemical differences. Values between 0.5–1.0 Å indicates small deviations often within the noise of simulation methods. RMSD $>$ 1.0 Å suggest significant structural differences that could impact chemical capabilities of bonding to the active site (pocket) of the protein. In our case, results in Figure 4.1 shows average RMSD across three datasets (random subset of respectively $40000, 50000, 20000$ ligands from the EXSCALATE's original dataset separated by molecular weight) strongly indicate that the mixed-precision approach preserves structural conformation. This mixed-precision strategy was mainly used in the local machine due to its gaming GPU, used for testing purposes. On HPC-grade machine instead we've always used double precision.

This approach however highlighted that swap to lower numerical accuracy in this case *does not* imply higher performance, regardless what is the hardware used. The advantage of using single precision with lower-grade hardware comes at the side effect of modeling a **more constrained PES**. Being the hessian matrix built from differences of coordinates between steps of the minimization, using lower accuracy implies a less accurate PES, hence, making harder times for the minimization algorithm to fall into a minimum.

Timing the execution of the pipeline with the same SMILES subsets used for analysis; each subset dataset minimized 10 times per version, *mixed-precision* exhibited *slower* throughput with respect to using double precision also for coordinates. Respectively *1%* for first two datasets and *2.5%* slower for the heavier molecule dataset, at any significant

confidence level (one tail t-test *p-value* $<$ *2.2e-16*).

Establishing the quality of obtained minimized conformation molecules is not an easy task; assumptions on the problem were carried out to understand which method yields more stable conformers. Energetically, ligands obtained with mixed-precision showed in average lower energy, but not statistically different from the double precision approach (one tail t-test *p-value* = *0.762*, same procedure for RMSD calculation). Moreover, obtained lower *computed* energy might be an artifact of numerical imprecision obtained by single precision.

The conclusion we end up with for the final implementation is to use double precision for every operation, regardless of the GPU at disposition.

4.3.2. Result of optimizing non bonded interactions

The implemented optimization to reduce *atomic* operations and non coalesced accesses target ligands with more than 32 atoms, including hydrogen atoms. We first analyze the sizes of ligands of the EXSCALATE dataset.

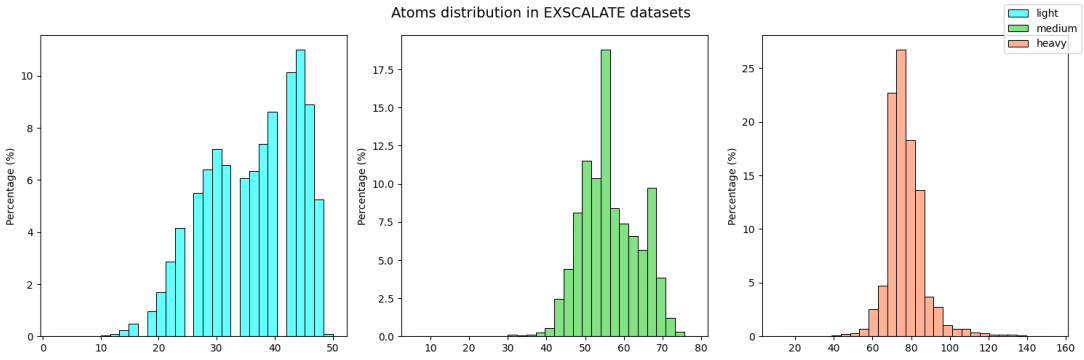
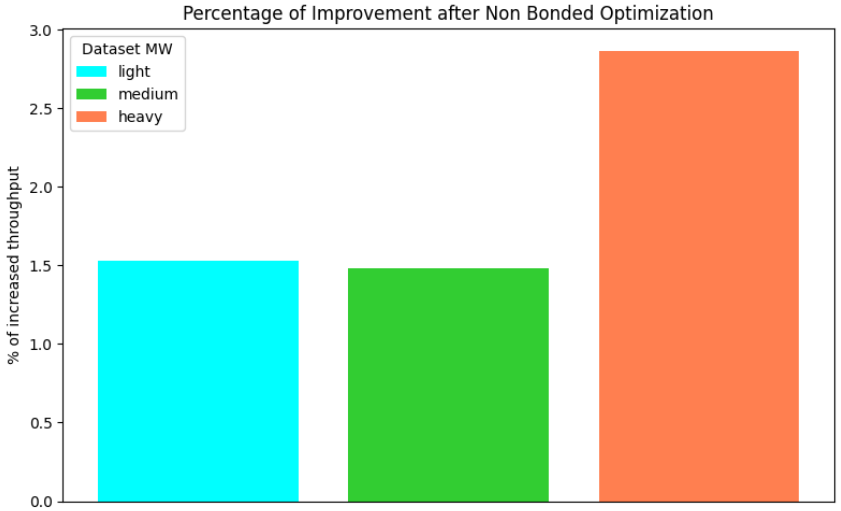


Figure 4.5: Atom distribution for the three EXSCALATE datasets.

Most of ligands have more than 30 atoms at a first glimpse. Quantifying, in *light* dataset 70.38% of ligands have more than 32 atoms, while for other two dataset almost 100% of ligands. This means that our optimization target most of the ligands in the dataset.

Moreover, in kernel we must allocate statically the number of registers: threads in prefetching must store their row and column atom's coordinate. This amount must be fixed because CUDA must allocate register per threads before kernel launching. Even if the dataset shows no bigger molecules, we assumed molecules with maximum $m = 196$, that may occurs as outliers in other datasets. However, this is an hyper parameter of the optimization proposed, that can be fixed depending on the dataset's property.

We executed the pipeline with and without this optimization and measured the throughput difference for each dataset and the percentage of improvement with respect to original one.



[H]

Figure 4.6: Percentage of increased throughput gained by optimizing non bonded Interaction using register prefetching.

Profiling with Nsight Compute the non-optimized version, target atom accesses exhibited a 12% inefficiency in global memory transactions due to non coalesced pattern, as well as gradient vector accesses showing same percentage. Profiling with the same dataset the optimized version, target atoms are accessed from global memory only in pre-fetching phase; the percentage of inefficiency was reduced down to 0.16%. While the optimization factor is considerable, is not reflecting in the same scale the reduction of uncoalesced accesses. This may be explained by a slight increase of computation that the optimization introduce; the non optimized implementation considered only interactions between atoms separated by at least three bonds, now instead we consider all pairs using dummy interactions in that case. Moreover, the synchronization of thread in warp when sharing target coordinates ends up taking 1.43% of warp stall sampling; this synchronization factor was not present in the naive implementation.

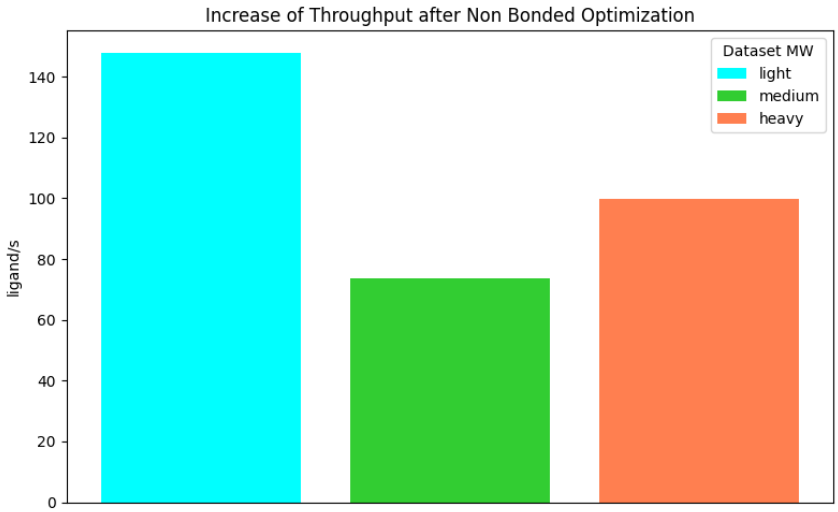


Figure 4.7: Similarly we show the amount of increased throughput after the non bonded optimization.

Figure 4.7 provides the benefit factor in terms of throughput with respect to the version without non bonded optimization. That is, in one second we respectively produce 148, 76 and 98 ligands more than the non optimized version.

4.3.3. Bucketizer Strategy Analysis

Minimizing the work imbalance of blocks has been both an intuitive optimization we immediately had in mind (the longest ligand will slow all ligands in the same wave) but also suggested by Nsight Compute profiling. We've used for profiling only the subset of 20000 ligands. After having optimized the non bonded interactions, the software suggested that also work imbalance optimizations would bring up a kernel optimization by 14% for the *light ligands dataset*. Even though this number is retrieved from a sampling, and is assuming no changes to the code are done, the suggestion was followed.

Rather than an optimization, we define the bucketizer strategy as a minimization of side effects (work imbalance). This because is very hard to predict a priori the complexity of minimization of a ligand, since depends on a vast amount of factors. We modeled the expected complexity of the ligand's PES with only one feature (number of *rotable bonds*) and due to the lack of dataset, we can't cope with this (is unimaginable to get the conformational entropy of such a vast amount of ligands, and hence train a model for this). Moreover, the need of a model very fast at inference reduced the choice to simple alternatives, discarding Neural Nets or tree based algorithm like boosting.

However, we prove that the direction followed in the thesis is promising. While for heavier

molecules we have no significative statistical evidence that the throughput of the kernel is increased with the strategy, we have instead for small ligand's dataset, at any significative confidence level, an increased throughput. We have confidence at 10% for medium weight ligands but not at 5%.

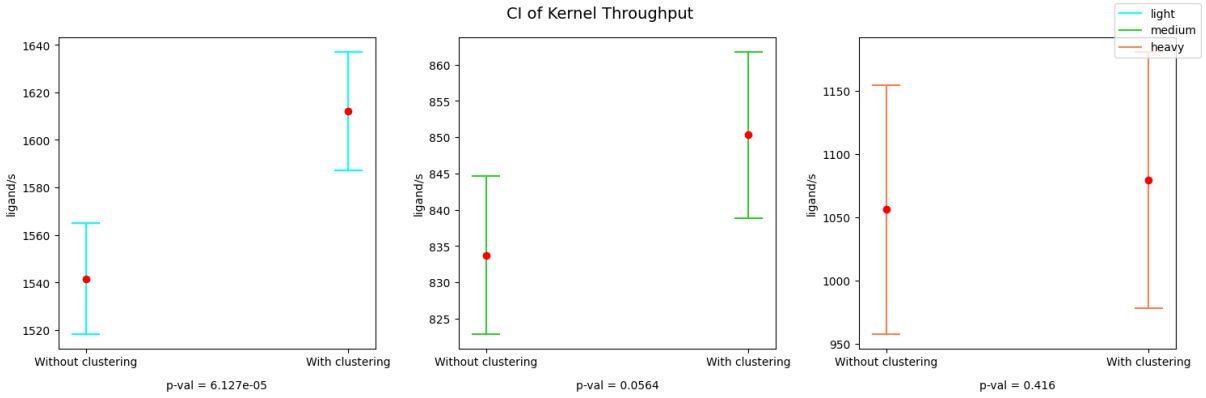


Figure 4.8: Confidence interval of kernel throughput at 90%. P-value of one-tailed t-test against pipeline without strategy.

The clustering by rotable bonds, however, depends on the dataset, the heavier dataset is at the same time the smaller one; we tried to interpret this result by assuming is due to lower variability in terms of entropy, making the strategy less effective. At the same time, the overall order is not altered in this clustering strategy, but only the intra-bucket one. The impact of the total ordering of SMILES wasn't carried out in this work, even if we expect it impacts also the intra-bucket clusterization structure, since is limiting the possible ligands in the bucket.

We highlight that those results are obtained even with a rudimentary incremental clustering model, without any policy of rearrangement before pushing to minimizer phase once all ligands are clusterized.

Figure 4.9 provides the throughput difference of the *entire pipeline* between version with the strategy and without, since the end-to-end throughput is the main interest. The application is then profiled again with Nsight Compute on the same subset previously used and the suggestion about the work imbalance showed a lower percentage (8%) of further optimization with respect to the original 14%, and we didn't change any kernel code for this.

H

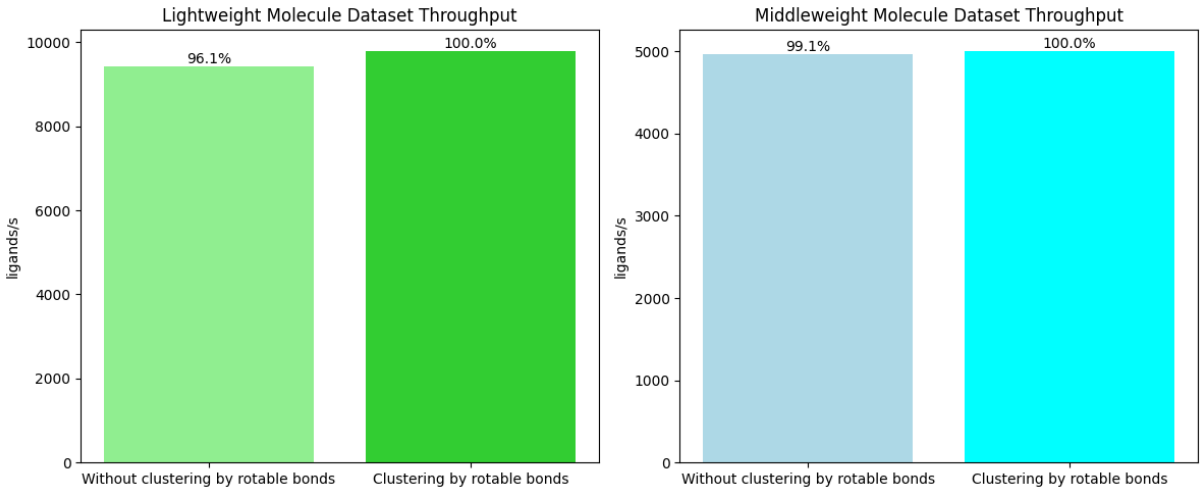


Figure 4.9: Throughput difference of the pipeline with respect to not using any bucketizer strategy for light and medium sized molecules.

4.4. Speedup CPU only vs CUDA integration

Being the final application hybridly parallelized with both CPU and GPU, the end-to-end throughput depends on the combination of allocated threads and GPU's power. The more the GPU power is available, the more the bottleneck of the application shifts toward other stages than the minimizer, dictating the throughput of the pipeline. Bottleneck's stage alternates between *guesser* and *minimizer*. Since the *minimization* phase is the most computationally intensive one, we always tried to shift the bottleneck to that phase. Moreover, GPU's price are order of magnitude higher than CPU's, so we desire to use them always at full capacity.

Equivalently to the procedure carried out in Section 3.2.1, also the pipeline with the kernel's implementation for minimizing buckets of ligands must be tuned. First, the pipeline has been tested on the local machine, equipped with a gaming GPU. The same thread configuration for the stages than the CPU's only pipeline is not appropriate since the system only has one GPU. In Figure 4.2 is provided the speedup of the sole minimization algorithm on CPU with the kernel one. As expected, the bigger the ligands, the bigger the speedup compared to the CPU serial code.

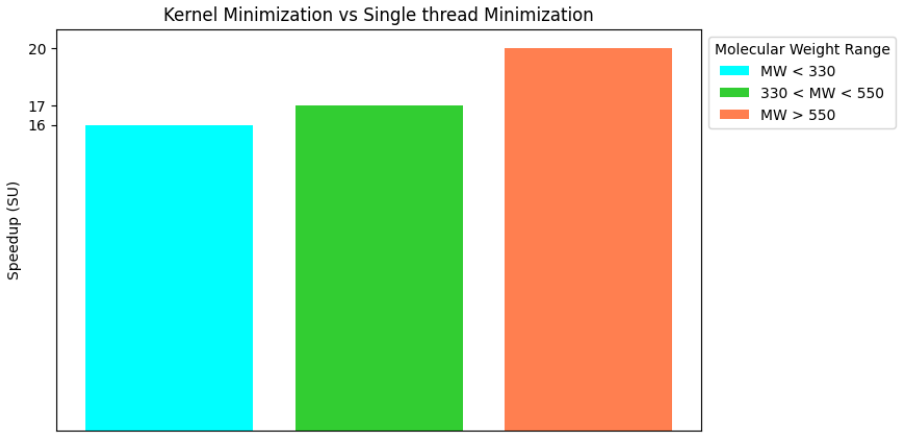


Figure 4.10: Speedup of the minimization algorithm for a bucket of ligands with respect to the same algorithm, serially, on CPU.

Remark: this is *not* the speedup of the two pipelines, but only of the minimization algorithm of buckets of ligands with respect to the baseline serial code of LiGen. So this result must not be interpreted as the final speedup obtained but is necessary to understand that, after porting the minimization to kernel, using the same number of cores for the guesser stage is not anymore enough to keep up with the new minimizer's throughput; indeed, using the same configuration of the CPU pipeline with the kernel's minimizer, the throughput of the pipeline is dictated by the initial guess stage.

At the previous configuration for which results of Figure 4.2 was obtained, we allocated 8 threads for guesser, 2 for minimizer, 1 for reader and 1 for bucketizer. By duplicating the threads of the guesser, we make the minimizer the bottleneck again, but only for the gaming GPU. Using an HPC-grade GPU doubling the threads of the guesser was still not enough. In Figure 4.11 is provided the pipeline's queue evolution on the *accento* with same configuration used on *local* machine.

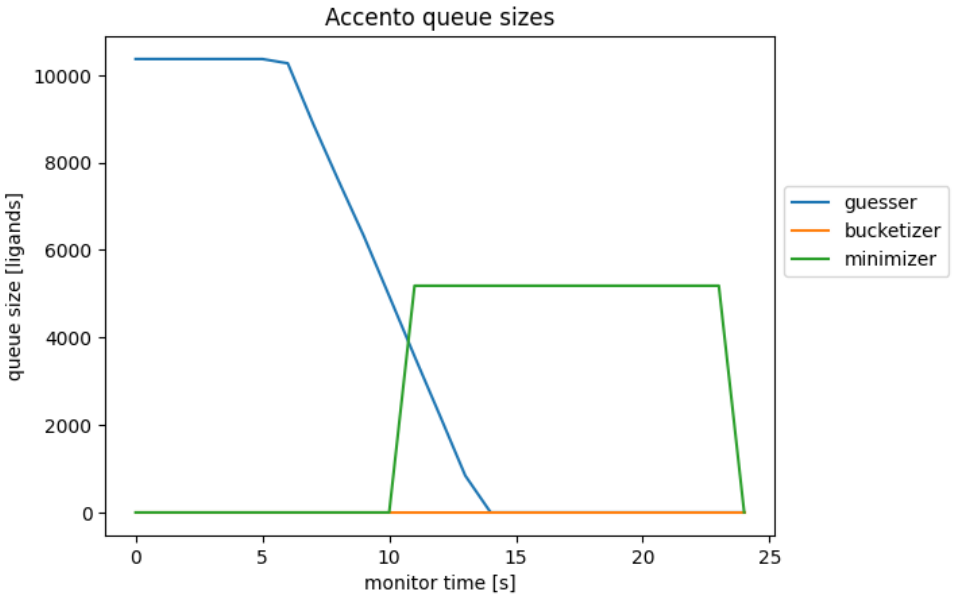


Figure 4.11: Bottleneck is not the minimizer anymore, but the *guesser* stage.

The procedure of bottleneck identification (with subsequent increase of its parallelization) must be carried out every time we execute the pipeline on a different architecture since this scenario of bottleneck shift may occurs, we must identify this opportunity of throughput increase.

Threads that with the CPU version were previously allocated to the minimizer now get allocated to the guesser, thanks to the porting on GPU of the minimization algorithm. This shift of thread allocation is always done with the tuning explained in Section 3.2.1, and the procedure is shown in Figure 4.12, conceptually equivalent with Figure 4.2. Comparing the two results, the trend obtained is a rotation of configuration, visually captured by the fact that colors in CPU parallelism scaled in the direction of rows for first configurations, now they scale in the direction of column in the figure below. Top-right corner indicates the optimum configuration zone, highlighting the need of shifting to at least 54 threads to overcome the bottleneck of the guesser. Similarly, we've allocated all physical cores and then hitting the wall. This means that our application scales well with CPUs cores.

Speedup is calculated with respect to the serial code version, since different allocations are performed than the parallel only CPU.

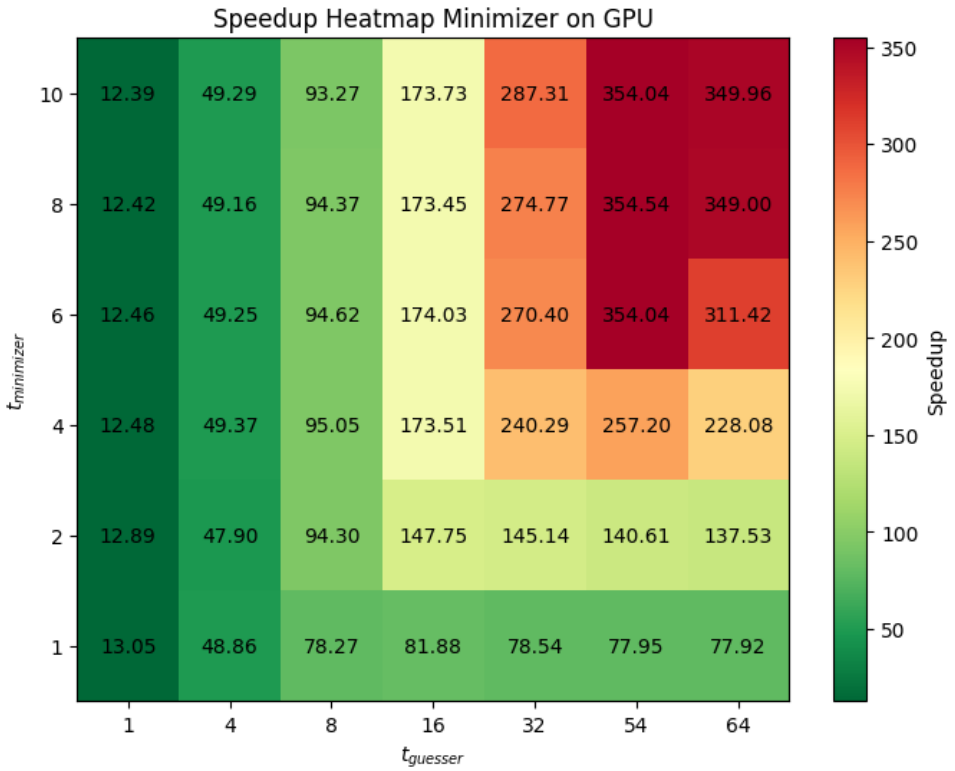


Figure 4.12: Speedup variation in function of threads allocation, after porting the minimizer to GPU.

Table 4.2 reports the configuration of threads for each stage for each available machine with both versions to further highlight the swapping of thread stages allocation.

Table 4.2: Threads of the fine-tuned pipelines on different machines. [*] *accento* has 20 cores in total. With this configuration we allocated all available threads, even though the bottleneck is the guesser. For this reason, CPU version only wasn't tested on that machine and speedups aren't provided.

Stage	**Algorithm version**	**Local**	**Accento***	**LIGATE**
Reader	CPU/GPU	1/1	-/1	1/1
Guesser	CPU/GPU	8/16	-/16	6/54
Bucketizer	CPU/GPU	1/1	-/1	1/1
Minimizer	CPU/GPU	8/2	-/2	56/8

In Table 4.3 are instead reported the throughput of the two versions of the pipeline on the target machine.

Table 4.3: Throughput and speedup of the fine-tuned pipelines version on LIGATE, with EXSCALATE datasets, against optimized parallelization on CPU.

Version	**Light**	**Medium**	**Heavy**
CPU only [$ligand/s$]	2851.888	1021.602	496.374
With GPU [$ligand/s$]	9800.655	5063.632	3590.818
Speedup	**3.437**	**4.957**	**7.234**

Kernel minimization time for buckets of very light molecules is measured to be very low, thus the overhead of launching the kernel is likely the limiting factor of bigger speedups. The latency of the kernel launches is likely hidden the more the kernel times increase (the minimization takes longer for bigger ligands). The provided speedup are still restricted by the available cores of the LIGATE machine. The fine tuning procedure on this machine drove us to a thread configuration where the guesser is dictating the throughput of the pipeline for some part of the computation. If we had more cores available we'd be able to make the minimizer always the one who establish the throughput, probably ending up with a bigger speedup factor.

Moreover, being the ported application a part of a software that is currently using GPU for the docking phase, by minimizing buckets of ligands on GPU we have the advantage that no further allocation and copies to device must be done, since ligands are already present in device memory.

5 | Conclusions

Computational methods in drug discovery mitigated main challenges such as high costs and required time. With Virtual Screening campaigns researchers can now assess potential drug candidates in-silico before conducting extensive laboratory experiments. A broader search increases the chances of identifying effective drug candidates, making in-vivo screening more targeted. This lead to increasing libraries of compounds over the years. One new computational challenge risen in VS campaigns is an efficient storage of such vast amount of ligands. Compact formats like the SMILES are significantly reduce the storage requirements compared to molecular coordinates. At the same time, docking algorithms need molecule's coordinate at stable conformation. The proposed method in this thesis aim to reduce the time needed to expand the molecules from compact format (SMILES) to low-energy conformations, starting from a state-of-the-art application: *LiGen-pre*. First the software was dissected and analyzed to understand its workflow and main computational challenges (that ended up being the minimization algorithm). For both the workflow and the algorithm, even though with different granularity, parallelization techniques are employed in order to scale up. Therefore, any hardware available in HPC infrastructures was considered, introducing parallelism with both CPU and GPU. A more heterogeneous parallelism is implemented with processors, that executes stages of the workflow in parallel, while at a lower resolution, the minimization algorithm for sets of ligands was ported to GPU. Challenges of fully exploiting the GPU hardware was faced with reorganization of computations and heuristic-based optimization, whose results were shown in Chapter 4. The proposed solution proved to increase the throughput of the original application without any loss in quality of generated ligands, with the main power of its ability to scale to any hardware available. Moreover, at our knowledge, optimizing energy minimization of high quantity of ligands is still a problem that was not currently faced in industry or research.

This work can be expanded in several ways; the same schema of energy evaluation can be evaluated for any system, not only small ligands. Non bonded interactions are typically evaluated also for proteins, that have order of magnitude more atoms that ligands. Since speedup suggests that big ligands benefited with the introduced schema, it should be

investigated the effect also for proteins. Moreover, the same schema can be used to remove accesses also for interactions parameters: rather than using interaction parameters, atom's parameters of the interaction could be considered and then retrieve the interaction parameters by means of *mixed-rules*. One challenge of this approach may be the handling of dummy interactions to avoid numerical issues. Alternatively, other force fields that consider all atom pairs should be taken in consideration. The bucketizer strategy is a starting point, and more advanced clustering techniques should be studied. Finally, the implementation of this thesis used CUDA for the kernel. To increase the portability of the application other technologies should be explored, such as SYCL or HIP.

Bibliography

[1] G. Accordi, D. Gadioli, E. Vitali, L. Crisci, B. Cosenza, A. Beccari, and G. Palermo. Out of kernel tuning and optimizations for portable large-scale docking experiments on gpus. *The Journal of Supercomputing*, 80(3):11798–11815, 2024. doi: 10.1007/s11227-023-05884-y. URL https://link.springer.com/article/10.1007/s11227-023-05884-y.

[2] P. T. Boggs and J. W. Tolle. Adaptive, limited-memory bfgs algorithms for unconstrained optimization. *SIAM Journal on Optimization*, 28(1):462–490, 2018. doi: 10.1137/16M1065100. URL https://doi.org/10.1137/16M1065100.

[3] Boost C++ Libraries. Boost C++ Libraries, 2025. https://www.boost.org/.

[4] L. Chan, G. M. Morris, and G. R. Hutchison. Understanding conformational entropy in small molecules. *Journal of Chemical Theory and Computation*, 17(4):2099–2106, 2021. doi: 10.1021/acs.jctc.0c01213. URL https://doi.org/10.1021/acs.jctc.0c01213. PMID: 33759518.

[5] N. Corporation. Nvidia nsight compute, 2025. URL https://developer.nvidia.com/nsight-compute. Accessed: 2025-02-11.

[6] Dompé farmaceutici S.p.A. Exscalate. https://www.dompe.com/it/scienza/exscalate/. Accessed: 2025-02-07.

[7] P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, et al. Openmm 7: Rapid development of high-performance algorithms for molecular dynamics. *PLOS Computational Biology*, 13(7):e1005659, 2017. doi: 10.1371/journal.pcbi.1005659.

[8] J. P. Ebejer et al. Journal of chemical information and modeling. *J. Chem. Inf. Model.*, 52(5):1146–1158, 2012.

[9] D. Gadioli, E. Vitali, F. Ficarelli, C. Latini, C. Manelfi, C. Talarico, C. Silvano, C. Cavazzoni, G. Palermo, and A. R. Beccari. Exscalate: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight sars-cov-

2. *IEEE Transactions on Emerging Topics in Computing*, 11(1):170–181, 2023. doi: 10.1109/TETC.2022.3187134.

[10] M. Garland and D. B. Kirk. Understanding throughput computing. In *GPU Technology Conference (GTC)*, 2010. URL `https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf`.

[11] R. Guha. *Chemical Data Mining: Challenges and Opportunities in Big Data*. Springer, 2014.

[12] I. D. Kuntz, J. M. Blaney, S. J. Oatley, R. Langridge, and T. E. Ferrin. A geometric approach to macromolecule-ligand interactions. *Journal of Molecular Biology*, 161 (2):269–288, 1982. doi: 10.1016/0022-2836(82)90153-X.

[13] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

[14] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry*, 32(10):2319–2327, 2011. doi: 10.1002/jcc.21787.

[15] C. F. Morris and J. M. Irwin. Assessing docking accuracy: Rmsd thresholds and their impact on success rates. *Journal of Enzyme Inhibition and Medicinal Chemistry*, 31 (3):456–472, 2016. doi: 10.1080/14756366.2016.1193736.

[16] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York, 2 edition, 2006. ISBN 978-0-387-30303-1. doi: 10.1007/978-0-387-40065-5. URL `https://doi.org/10.1007/978-0-387-40065-5`.

[17] NVIDIA Corporation. CUDA C Programming Guide. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2023. URL `https://www.openmp.org/specifications/`.

[19] S. M. Paul, D. S. Mytelka, C. T. Dunwiddie, C. C. Persinger, B. H. Munos, S. R. Lindborg, and A. L. Schacht. How to improve r&d productivity: the pharmaceutical industry's grand challenge. *Nature Reviews Drug Discovery*, 9(3):203–214, 2010.

[20] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005. doi: 10.1002/jcc.20289.

[21] RDKit: Open-Source Cheminformatics Software. Rdkit: Open-source cheminformatics, 2024. https://www.rdkit.org/.

[22] B. K. Shoichet. Virtual screening of chemical libraries. *Nature*, 432, 2004.

[23] K. Takaba, A. J. Friedman, C. E. Cavender, P. K. Behara, I. Pulido, M. M. Henry, H. MacDermott-Opeskin, C. R. Iacovella, A. M. Nagle, A. M. Payne, M. R. Shirts, D. L. Mobley, J. D. Chodera, and Y. Wang. Machine-learned molecular mechanics force fields from large-scale quantum chemical data. *Chemical Science*, 15(32):12861–12878, 2024. ISSN 2041-6539. doi: 10.1039/d4sc00690a. URL http://dx.doi.org/10.1039/D4SC00690A.

[24] O. T. Unke, S. Chmiela, H. E. Sauceda, M. Gastegger, I. Poltavsky, K. T. Schütt, A. Tkatchenko, and K.-R. Müller. Machine learning force fields. *Chemical Reviews*, 121(16):10142–10186, 2021. doi: 10.1021/acs.chemrev.0c01111. URL https://doi.org/10.1021/acs.chemrev.0c01111. PMID: 33705118.

[25] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. C. Berendsen. Gromacs: Fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, 2005. doi: https://doi.org/10.1002/jcc.20291. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20291.

[26] P. K. Weiner and P. A. Kollman. Amber: Assisted model building with energy refinement. a general program for modeling molecules and their interactions. *Journal of Computational Chemistry*, 4:236–247, 1984. doi: 10.1002/jcc.540050204.

[27] D. Weininger. SMILES 1. introduction and encoding rules. *Journal of Chemical Information and Modeling*, 28, 1988.

List of Figures

2.1 A ligand fitting in one of the binding site of a protein 6
2.2 Cyclohexane (SMILES: C1CCCCC1) molecule with a ring of 6 Carbon atoms 7
2.3 Typical procedure to build empirical force field 9
2.4 Morse potential 11
2.5 Wilson angle 13
2.6 Torsional angle from "Effect of doped indium" by Muhaj Talib et. al. . . . 14
2.7 Torsional contribution energy to ethane 15
2.8 Van der Waals potential 17
2.9 LiGen-pre workflow 24
2.10 Automatic scalability [17] 27
2.11 CUDA hierarchy 29
2.12 Must use registers to run close to the peak, this require many registers, that translates in lower occupancy. [10] 30

3.1 Ligen stages parallelized, organizing computations in a pipelined architecture. 34
3.2 Correlation matrix of k (iteration number to reach convergence) and ligand features. 44
3.3 GPU kernel map of computation. One block minimize a ligand of the bucket. One general interaction "inter" is represented. In implementation, there are several arrays like this, once per interaction. 47
3.4 Prefetching atom coordinates, storing coordinate values in register. Source and target atom handled by thread are identified respectively by the left column vector and bottom row vector, with the responsible thread encoded by its color. 53
3.5 Reorganization of non bonded interaction matrix. Each warp is identified by the main color, than each thread inside the warp is identified by a its warp color, but with different nuance. With this picture we aim to show how target atom is shared among threads from previous iteration gradient evaluations. 55

3.6 Warp iterations and sharing pattern after each interaction evaluation. Using the same color of the interaction matrix to show the order of processed interactions. 56
3.7 On the right we see an example of leftovers interactions handled using **column reduce pattern**. Same color encoding with shades used in previous figure. 58

4.1 Pipeline's stages queue size in time. Queue size of each stage logged every second. 64
4.2 Speedup variation depending on pipeline configuration. 65
4.3 Optimal configuration queue evolution. 66
4.4 Confidence interval for RMSD between ligands minimized using float and double coordinates. 67
4.5 Atom distribution for the three EXSCALATE datasets. 68
4.6 Percentage of increased throughput gained by optimizing non bonded Interaction using register prefetching. 69
4.7 Similarly we show the amount of increased throughput after the non bonded optimization. 70
4.8 Confidence interval of kernel throughput at 90%. P-value of one-tailed t-test against pipeline without strategy. 71
4.9 Throughput difference of the pipeline with respect to not using any bucketizer strategy for light and medium sized molecules. 72
4.10 Speedup of the minimization algorithm for a bucket of ligands with respect to the same algorithm, serially, on CPU. 73
4.11 Bottleneck is not the minimizer anymore, but the *guesser* stage. 74
4.12 Speedup variation in function of threads allocation, after porting the minimizer to GPU. 75

List of Tables

3.1 Reported linear model obtained in reference paper "Understanding Conformational Entropy in Small Molecules" [4] with the ligand's feature in first column. 43
3.2 Table defining interactions, atoms involved, and number of parameters. . . 48
3.3 Global memory access to gradient and position of the new strategy compared to the naive, row-major organization of interaction matrix. The without strategy accesses are approximated values since real values will be slightly lower, since are considered only atoms separated by at least three bonds. With strategy values are instead exact, because of dummy interactions. 54

4.1 Performance profiling data for the serial original LiGen application. 63
4.2 Threads of the fine-tuned pipelines on different machines. [*] *accento* has 20 cores in total. With this configuration we allocated all available threads, even though the bottleneck is the guesser. For this reason, CPU version only wasn't tested on that machine and speedups aren't provided. 75
4.3 Throughput and speedup of the fine-tuned pipelines version on LIGATE, with EXSCALATE datasets, against optimized parallelization on CPU. . . 76

Acknowledgements

Voglio ringraziare prima di tutto il dipartimento DEIB, in particolare il Prof. Gianluca Palermo, Davide Gadioli e Gianmarco Accordi per la loro continua disponibilità, per avermi guidato durante lo svolgimento della tesi e per i caffè necessari per la stesura del codice. Porterò i loro consigli con me d'ora in poi per crescere ulteriormente a livello professionale.
Un enorme grazie va i miei genitori, per ogni tipo di sostegno dal primo anno di vita a questo momento, sia personale, che educativo, che economico. Senza di voi non sarei sicuramente quello che sono ora, e per questo ve ne sarò sempre grato.
A mio padre per avermi insegnato a vivere nel mondo reale e a non prendere niente sotto gamba, per tutti i discorsi di ore e ore in macchina e le giornate al freddo e al gelo.
A mia madre per ogni gesto di conforto dopo ogni delusione o semplicemente per ricarica a inizio giornata.
Ringrazio mio fratello per ogni consiglio essendo passato prima di me su questa strada, trasferendomi tutta la sua esperienza, e per ogni "stai tranquillo" quando tranquillo non lo ero. Questo mi ha permesso un vantaggio sui tempi che non avrei avuto senza di te.
Ringrazio Veronica per ogni "in bocca al lupo" alle 6 del mattino prima di un esame.
A mia nonna, per le pizze, i gelati e i panzerotti mangiati insieme dopo le giornate passate a studiare.
Un grazie speciale alla mia ragazza, che nei suoi piccoli gesti quotidiani mi ha aiutato ad affrontare il mio percorso magistrale serenamente, aiutandomi a superare le difficoltà lungo il cammino. Sia venendo da me per ottimizzare i tempi, sia alla sua semplice presenza e sapere di averla affianco.
Infine ringrazio i miei compagni di corso, che mi hanno permesso di dividere il carico soltanto condividendo un pasto insieme o semplici momenti no-brain.

Grazie mille a tutti voi, che insieme siamo arrivati qui.