Analysis and Implementation of Methods for Identifying Target Sites within Proteins to Support Drug Discovery

Niccolò Benetti Lorenzo Trombini

August 18, 2025

Contents

1 Introduction **3**
- 1.1 Participants . . . 3
- 1.2 Contents . . . 3
- 1.3 Tools . . . 4
  - 1.3.1 Development Environment . . . 4
  - 1.3.2 Performance Profiling . . . 4
  - 1.3.3 Collaboration . . . 5

2 Development Process **6**
- 2.1 Phase 1: Learning and Preparation . . . 6
- 2.2 Phase 2: Non parallelized code development . . . 7
  - 2.2.1 Design and Planning . . . 7
  - 2.2.2 Implementation . . . 7
- 2.3 Phase 3: Hardware Acceleration and Optimization . . . 7
  - 2.3.1 CUDA Integration . . . 8
  - 2.3.2 Performance Profiling and Optimization . . . 8
  - 2.3.3 Data Collection and Analysis . . . 8

3 Project Description **9**
- 3.1 Sequential workflow (CPU only) . . . 9
  - 3.1.1 Overview . . . 9
  - 3.1.2 Output file . . . 11
  - 3.1.3 Data Structures . . . 12
  - 3.1.4 Modeled Intermolecular Interactions . . . 13
  - 3.1.5 SMARTS Patterns . . . 15
- 3.2 Parallelized workflow (GPU acceleration) . . . 16
  - 3.2.1 Why GPU acceleration here . . . 16
  - 3.2.2 Data Preparation and Memory Management . . . 17
  - 3.2.3 Asynchronous execution . . . 17
  - 3.2.4 Block sizes and grid layout . . . 18
  - 3.2.5 Numerical strategy . . . 18

3.2.6 Interaction-Specific Kernels 18
3.2.7 Host–Device Interface and Output 20

4 Results **21**
4.1 Tools to compare the implementations 21
4.2 The GPU performance tradeoff 23
4.2.1 Overview 23
4.2.2 Where GPU overheads come from 24
4.2.3 When GPUs outperform CPUs 24
4.2.4 What we observed in our pipeline 24

5 Innovation of the Project **29**
5.1 Key Innovations 29
5.1.1 Enhanced Computational Efficiency 29
5.1.2 Scalability 29
5.2 Impact of the Innovation 30

6 Project Setup Guide **31**
6.1 Prerequisites 31
6.2 Build Instructions 31
6.3 Running the Program 32
6.4 Testing and Utility Scripts 32
6.5 How to generate the complete code documentation with Doxygen 33

7 Conclusions **34**
7.1 Future Work 34
7.2 Final Thoughts 35

Chapter 1

Introduction

1.1 Participants

The project was conducted by Niccolò Benetti and Lorenzo Trombini. Throughout the development of this project, we were closely guided and supported by Professor Gianluca Palermo, Researcher Davide Gadioli, and Researcher Gianmarco Accordi.

Their expertise and insights were invaluable in ensuring the project's success, particularly in areas requiring specialized knowledge in computational methods and molecular interactions. Their contributions ranged from providing initial project direction to offering feedback during the development and testing phases.

1.2 Contents

The project involves developing a C++ code that is later accelerated using CUDA to enhance performance. The primary goal of the code is to analyze a given protein and several ligands to identify potential interactions. This is achieved by leveraging the RDKit library, which provides powerful tools for cheminformatics, particularly in analyzing molecular structures and interactions. The accelerated code allows for more efficient processing, making it feasible to handle larger datasets or more complex calculations within a shorter timeframe.

The findings from this project can be highly beneficial in several fields, including drug discovery, where understanding protein-ligand interactions is crucial for identifying potential drug candidates. Additionally, it can be applied in bioinformatics for predicting binding sites and in computational chemistry for studying molecular interactions at a detailed level.

1.3 Tools

Throughout the development of this project, we utilized a variety of tools and technologies to facilitate collaboration, development, and performance analysis.

1.3.1 Development Environment

The project was developed in C++, with different tools used during the various stages of development:

- **GCC Compiler**: During the initial, non-accelerated phase, we used the GCC compiler to compile our C++ code. This allowed us to test and verify the functionality before implementing hardware acceleration.
- **CUDA and NVCC Compiler**: For the hardware-accelerated phase, we programmed in **CUDA**, a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs. The **NVCC compiler** was used to compile the CUDA code, enabling us to leverage the computational power of NVIDIA GPUs.
- **RDKit**: For molecular analysis and pattern matching, we used RDKit, an open-source cheminformatics toolkit. It allowed us to parse molecular structures in PDB and Mol2 formats (while retaining hydrogen atoms), construct query molecules from SMARTS patterns, identify substructures within target molecules and complete many other.
- **CMake**: To manage the build process in a portable and modular way, we used CMake as our build system.

1.3.2 Performance Profiling

To analyze and optimize the hardware performance of our CUDA-accelerated code, we used NVIDIA's profiling tools, including:

- **Nsight Systems**: This tool provided detailed insights into the performance of our CUDA code, helping us identify bottlenecks and optimize the execution time. It was instrumental in ensuring that our application made full use of the GPU's capabilities.

These tools collectively supported the successful completion of the project, from initial development to final optimization, providing us with both the technical means and collaborative environment necessary for our work.

1.3.3 Collaboration

For version control and collaborative development, we used **GitHub**. This platform allowed us to manage our codebase effectively, track changes, and collaborate seamlessly, ensuring that all project members were synchronized.

Chapter 2

Development Process

The development of this project was structured into three main phases, each critical to the successful completion of the final product. These phases included initial learning and preparation, code development, and hardware acceleration with performance optimization. Below, we provide a detailed account of each phase.

2.1 Phase 1: Learning and Preparation

Before diving into the code development, we recognized the need to build a solid foundation in CUDA programming. To this end, we began by studying two essential courses provided by NVIDIA:

- **Getting Started with Accelerated Computing in CUDA C/C++**: This course served as our introduction to CUDA programming, covering the basics of writing, compiling, and running CUDA programs. It was instrumental in helping us understand the fundamentals of GPU architecture and parallel programming.
- **Accelerating CUDA C++ Applications with Concurrent Streams**: This course provided us with an in-depth understanding of how to manage and optimize multiple streams of execution within CUDA, a crucial skill for maximizing GPU utilization and improving performance in parallel computing environments.

These courses laid the groundwork for our subsequent work, ensuring that we had the necessary knowledge and skills to implement CUDA effectively in our project.

2.2 Phase 2: Non parallelized code development

With a solid understanding of CUDA and parallel computing, we moved on to the second phase: writing the initial C++ code for the project. This phase involved the following steps:

2.2.1 Design and Planning

We began by designing the overall structure of the application. The goal was to develop a C++ program capable of analyzing protein-ligand interactions using the RDKit library. We outlined the major components of the code, including data input/output, interaction analysis, and result processing.

2.2.2 Implementation

The implementation phase focused on writing the C++ code without hardware acceleration. This allowed us to build and test the core functionality of the program in a straightforward, CPU-based environment. The primary tasks included:

- Integrating the RDKit library to handle molecular data and perform interaction analysis.
- Developing algorithms for processing and analyzing protein-ligand interactions.
- Implementing file handling and data management functions to ensure smooth data flow through the program.

By the end of this phase, we had a working C++ application that could analyze protein-ligand interactions on a CPU, providing a baseline for further optimization.

2.3 Phase 3: Hardware Acceleration and Optimization

The final phase of the project was dedicated to hardware acceleration and optimization, where we leveraged the power of NVIDIA GPUs to enhance the performance of our C++ code.

2.3.1 CUDA Integration

We started by converting the most computationally intensive parts of our code to run on the GPU using CUDA. This involved:

- Identifying bottlenecks in the CPU code that would benefit most from parallelization.
- Rewriting key functions in CUDA to enable parallel processing on the GPU.
- Compiling and testing the CUDA code using the NVCC compiler to ensure it functioned correctly and efficiently on the GPU.

2.3.2 Performance Profiling and Optimization

Once the CUDA code was functional, we entered a cycle of profiling and optimization to further improve performance. Using NVIDIA's Nsight Systems, we conducted detailed profiling to identify any remaining inefficiencies in our GPU code. This allowed us to:

- Optimize memory usage and data transfer between the CPU and GPU, reducing latency and maximizing throughput.
- Fine-tune the parallel execution of tasks to fully exploit the computational power of the GPU.
- Iterate on the code, making incremental improvements based on profiling data to achieve the best possible performance.

2.3.3 Data Collection and Analysis

With the code finalized, we collected data on the performance of our application under various conditions. This data was then analyzed to assess the effectiveness of the hardware acceleration and to compare the performance of the GPU-accelerated version against the original CPU-based implementation.

The development process for this project was iterative and highly focused on optimizing performance through hardware acceleration. Each phase built upon the last, leading to a final product that is both powerful and efficient. The skills and knowledge gained through this process were invaluable, providing us with hands-on experience in high-performance computing and parallel programming, which are essential competencies in the field of computer engineering.

Chapter 3

Project Description

3.1 Sequential workflow (CPU only)

3.1.1 Overview

The project is a command-line tool to detect and catalog all relevant intermolecular interactions between a protein (PDB input) and one or more ligands (Mol2 input). Using RDKit for molecule handling and SMARTS-based pattern matching, it identifies atom-level features (the detailed list is shown in the table 3.1.5). Geometric criteria such as distances, angles, ring planarity and others (see the full list in section: 3.1.4) are then applied to confirm whether each candidate pair satisfies the spatial conditions required for an actual interaction.

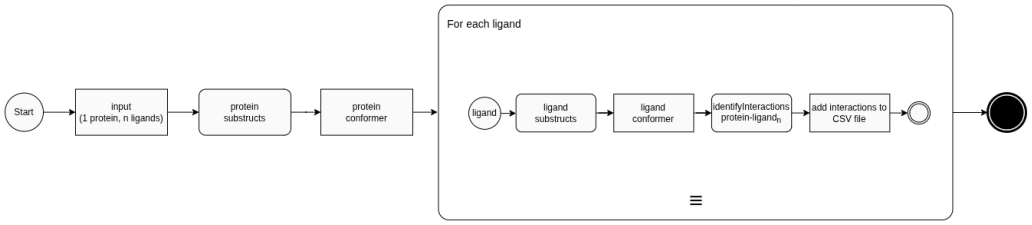

Figure 3.1: CPU only code workflow.

At the logical level, the program first locates all instances of each SMARTS pattern independently in the protein and in the ligand. For every interaction type, it then computes the Cartesian product of all matching features in the protein and in the ligand, and evaluates every resulting pair to check whether it meets the geometric constraints specific to that interaction. This ensures that all valid physical contacts are captured exhaustively.

The figure 3.2 is a representation of the complete workflow of the CPU-only program. The diagram shows the processes associated with the function responsible for the substructure identification (top) and the function that computes and identifies the interactions between the protein and the ligand (below).

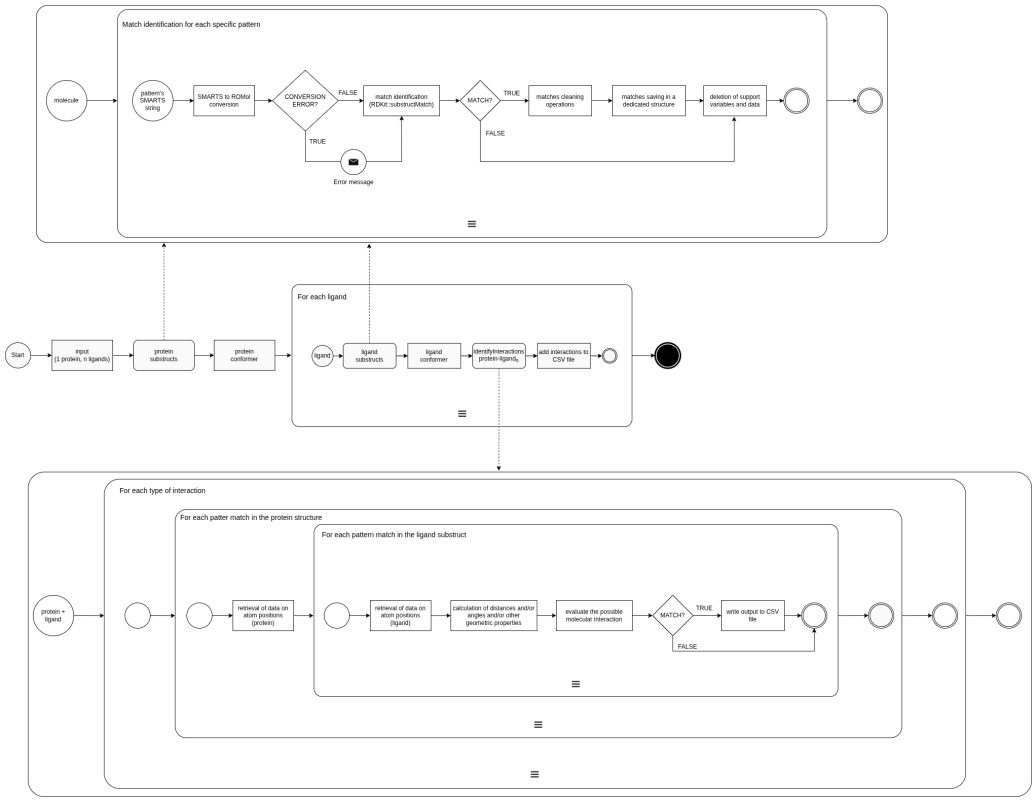

Figure 3.2: Complete CPU only code workflow.

3.1.2 Output file

The results are collected in a structured CSV file, where each row corresponds to a single interaction and includes identifiers, 3D coordinates, the assigned pattern roles, interaction type, and computed distance. An example of output data is given in the table below.

Table 3.1: Examples of data reported in the output CSV file

Field	Entry 1	Entry 2	Entry 3
Ligand name	./1a07_ligand	./1a07_ligand	./1a07_ligand
Protein atom ID	B.TYR205. CG	B.ARG158.2HH1	B.ARG158. NH2
Protein pattern	Hydrophobic	Hydrogen donor	Cation
Protein X	42.292	46.248	43.962
Protein Y	-2.907	-6.788	-6.23
Protein Z	44.092	36.183	35.465
Ligand atom ID	55(C)	37(O)	45(C)
Ligand pattern	Hydrophobic	Hydrogen acceptor	Aromatic_ring
Ligand X	44.13	44.733	40.3165
Ligand Y	-6.969	-7.088	-6.38583
Ligand Z	43.991	37.682	37.1967
Interaction type	Hydrophobic	Hydrogen Bond	Ionic
Interaction distance (Å)	4.45963	2.94185	4.03889

3.1.3 Data Structures

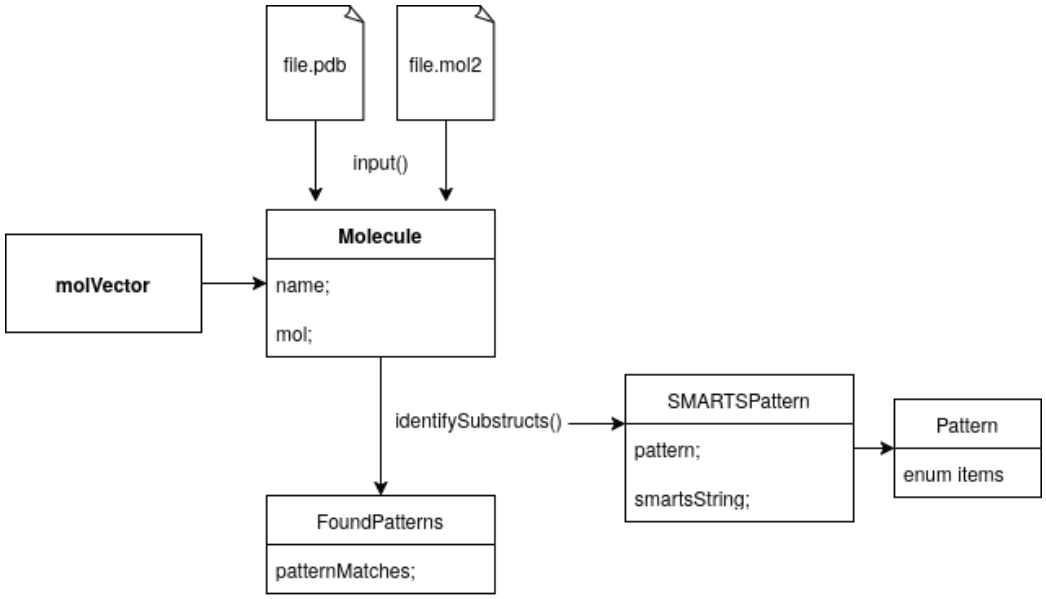

Figure 3.3

molVector Vector holding all parsed molecules: index 0 is the protein, and $1..n$ are ligands.

```
vector<Molecule> molVector;
```

Molecule A simple struct that pairs a name with an RDKit molecule:

```
struct Molecule {
    string name;
    unique_ptr<RDKit::ROMol> mol;
};
```

SMARTSPattern

Links a `Pattern` enum to its SMARTS query string.

```
struct SMARTSPattern {
    Pattern pattern;
    string smartsString;
};
```

FoundPatterns

Maps each pattern to the list of matched atom–index vectors.

```
struct FoundPatterns {
    map<Pattern, vector<RDKit::MatchVectType
        >> patternMatches;
};
```

Other RDKit Types

- `RDKit::Conformer` — Stores 3D coordinates
- `RDKit::ROMol` — immutable molecule class
- `RDKit::RWMol` — editable molecule class

3.1.4 Modeled Intermolecular Interactions

This section provides an overview of the types of intermolecular interactions that our program is designed to detect and analyze, along with the specific geometric constraints required for each interaction to be considered valid.

Notations

Throughout this documentation:

- Single atoms are denoted using square brackets, e.g., `[Hydrophobic]` refers to a hydrophobic atom.
- The character `-` represents an intramolecular bond.
- The character `...` represents an intermolecular interaction between two atoms from different molecules.

Typically, an interaction involves one atom from each molecule. In cases where more atoms are involved, the corresponding SMARTS pattern includes multiple atoms.

Intermolecular Interactions Detected

The following interactions are modeled and detected by the program:

Hydrophobic Interaction

Occurs between two hydrophobic atoms, represented as:

[hydrophobic] ... [hydrophobic]

Conditions:

- Atom-atom distance must be less than or equal to 4.5 Å.

Hydrogen Bond

Occurs between a hydrogen donor and a hydrogen acceptor, where the hydrogen atom is shared:

$$[\text{hydrogen\_donor}] - [\text{hydrogen}] \ldots [\text{hydrogen\_acceptor}]$$

Conditions:

- Distance between donor and acceptor $\leq$ 3.5 Å.
- Angle between donor-hydrogen-acceptor atoms between 130° and 180°.

Halogen Bond

Similar to hydrogen bonds but involves a halogen atom:

$$[\text{halogen\_donor}] - [\text{halogen}] \ldots [\text{halogen\_acceptor}] - [\text{any}]$$

Conditions:

- Distance between donor and acceptor $\leq$ 3.5 Å.
- Donor-halogen-acceptor angle between 130° and 180°.
- Halogen-acceptor-any angle between 80° and 140°.

Ionic Interaction

Involves either:

- A cation and an anion: $[\text{cation}] \ldots [\text{anion}]$
- A cation and an aromatic ring: $[\text{cation}] \ldots [\text{aromatic\_ring}]$

Conditions:

- For cation-ion interactions:
  - atom-atom distance $\leq$ 4.5 Å.
- For cation-aromatic ring interactions:
  - Distance to the ring centroid $\leq$ 4.5 Å.
  - Angle between the ring plane normal and the cation-centroid vector between 0° and 30°.

Pi Stacking Interaction

Occurs between two aromatic rings:

[aromatic_ring] ... [aromatic_ring]

Two possible configurations:

- **Sandwich stacking:**
  - Distance between centroids ≤ 5.5 Å.
  - Angle between planes 0° to 30°.
  - Angle between plane normal and centroid vector 0° to 33°.
- **T-shaped stacking:**
  - Distance between centroids ≤ 6.5 Å.
  - Angle between planes 50° to 90°.
  - Normal-centroid angle 0° to 30°.
  - The centroid of the perpendicular ring must project within the other ring.

Metal Coordination

Occurs between a metal atom and a chelated atom:

[metal] ... [chelated]

Condition:

- Atom-atom distance ≤ 2.8 Å.

3.1.5 SMARTS Patterns

Atom types are defined using SMARTS patterns. Below is a summary:

Pattern Name	Number of Atoms	SMARTS String
hydrophobic	1	[c,s,Br,I,S&H0&v2,$([D3,D4;#6])&!$([#6]~[#7,#8,#9])&!$([#6X4H0]);+0]
hydrogen_donor-H	2	[$([O,S;+0]),$([N;v3,v4&+1]),n+0]-[H]
hydrogen_acceptor	1	[#7&!$([nX3])&!$([NX3]-*=[O,N,P,S])&!$([NX3]-[a])&!$([Nv4&+1]),O&!$([OX2](C)C=O)&!$(O(~a)~a)&!$(O=N-*)&!$([O-]-N=O),o+0,F&$(F-[#6])&!$(F-[#6][F,Cl,Br,I])]
halogen_donor-halogen	2	[#6,#7,Si,F,Cl,Br,I]-[Cl,Br,I,At]
halogen_acceptor-any	2	[#7,#8,P,S,Se,Te,a;!+1-][*]
anion	1	[-1-,$(O=[C,S,P]-[O-])]
cation	1	[+1-,$([NX3&!$([NX3]-O)]-[C]=[NX3+])]
aromatic_ring	5 or 6	[a;r5]1:[a;r5]:[a;r5]:[a;r5]:[a;r5]:1 or [a;r6]1:[a;r6]:[a;r6]:[a;r6]:[a;r6]:[a;r6]:1
metal	1	[Ca,Cd,Co,Cu,Fe,Mg,Mn,Ni,Zn]
chelated	1	[O,#7&!$([nX3])&!$([NX3]-*=[!#6])&!$([NX3]-[a])&!$([NX4]),-1-;!+1-]

3.2 Parallelized workflow (GPU acceleration)

This chapter documents how CUDA based hardware acceleration is integrated into our C++/RDKit project for protein-ligand interaction analysis. The acceleration targets the computational hot spots where we evaluate large cartesian products of candidate interaction features. Each interaction type (hydrophobic, hydrogen bond, halogen bond, ionic, π–stacking, metal coordination) reduces to repeatedly computing geometry (distances, angles, planarity checks) under interaction specific thresholds. CUDA lets us run these independent pairwise evaluations in parallel on the GPU.

3.2.1 Why GPU acceleration here

It is convenient to transfer the computational work of evaluating all pairs $\langle A_i, B_j \rangle$ of feature instances found via SMARTS queries in the protein (A) and ligand (B) from the CPU to the GPU. The cost of this operation is

$O(|A|\cdot|B|)$ per interaction type, and several interaction types may be checked per ligand. CUDA reduces wall clock time by:

- mapping the 2D pair space to a 2D grid of threads,
- coalescing reads for one operand to maximize memory throughput,
- early out filtering before expensive operations (square roots / arccos),
- overlapping host–device transfers and kernel execution where feasible.

3.2.2 Data Preparation and Memory Management

RDKit pre-processing and feature extraction Proteins (PDB) and ligands (Mol2) are parsed with RDKit. We retain explicit hydrogens, run a light sanitization (aromaticity, conjugation, hybridization, ring info) and operate on `ROMol` conformers to retrieve 3D coordinates. Interaction relevant atom sets are detected via SMARTS patterns. The SMARTS detections are stored as match vectors per `Pattern`. For rings, we compute on the CPU both the centroid (average of ring atom positions) and a unit normal (cross product of two ring edges).

Pinned host memory and device buffers For high throughput copies, large staging arrays are allocated with page locked memory (`cudaMallocHost`). This is used in the interactions where we batch many pairs. Device arrays are allocated with `cudaMalloc`.

Transfer strategy To minimize traffic, data for the "B" operand is transferred once and reused across chunks of "A". For very large $|A|$, we split A into chunks and stream them (async) while kernels consume previously copied chunks. The default stream count is `NUM_STREAMS` (configurable). This technique is known as "Copy compute chunking". Where chunking is not required (or data is small), a single synchronous transfer is used for simplicity.

3.2.3 Asynchronous execution

When multiple streams are enabled, each stream handles:

1. async copy A-*chunk* $\rightarrow$ device,
2. kernel launch on that chunk,
3. async copy of the *result submatrix* back to host.

Finally, we synchronize the streams and perform a CPU pass over the result matrix to emit CSV rows for each validated interaction.

3.2.4 Block sizes and grid layout

`BLOCKSIZEX` and `BLOCKSIZEY` are compile-time constants that set how many CUDA threads each block contains along the **X** and **Y** axes. In our 2D mapping, **B** (ligand features) spans the grid's **X** axis and **A** (protein features) spans the **Y** axis. Each block therefore computes a small "tile" of the **A**×**B** result matrix:

- **`BLOCKSIZEX`** = threads per block that march across **B** (columns). Keeping this a multiple of 32 (warp size) helps coalesced reads from the **B** arrays.
- **`BLOCKSIZEY`** = threads per block that march across **A** (rows). We often keep this small to favor a wide **X** dimension and maximize memory throughput on **B**.

These settings control how work is tiled over the pair space and are the main levers to balance occupancy, memory coalescing and latency hiding for our distance/angle kernels.

3.2.5 Numerical strategy

Across kernels we apply the same principles:

- **Early-out on squared distances.** Compare d^2 to τ^2 before computing `sqrt`.
- **Cosine thresholds.** Angle windows are implemented via dot products and inverse norms; comparisons are made in cosine space to avoid premature `acos` calls.
- **Output encoding.** Invalid pairs write sentinel values (`-1`) so host-side post-processing can simply scan the dense matrix and pick positives.

3.2.6 Interaction-Specific Kernels

For clarity, all thresholds reported below are the defaults used in our build and are configurable in `main.hpp`. Distances are in Å and angles in degrees.

Hydrophobic interactions

Role mapping. A: hydrophobic atoms in protein; B: hydrophobic atoms in ligand.

Implementation. We stream chunks of A (pinned $\rightarrow$ device) while keeping B resident on device. The kernel computes the Euclidean distance and writes it if within the cutoff; otherwise it writes `-1`. Because the computation is memory-bound and trivially parallel, this kernel benefits most from coalesced B reads and especially large batches.

Hydrogen bonds

Role mapping. A: donors (and their bound H); B: acceptors.

Implementation. Each thread pulls donor, hydrogen, and acceptor triplets, applies a distance prefilter on D–A, then checks the D–H and H–A vectors via cosine threshold $\cos\theta \leq \cos(130^\circ)$. A compile-time flag can relax the geometrical check to distance-only if explicit hydrogens are missing in the input. Only positive pairs return a finite distance and therefore are accepted.

Halogen bonds

Role mapping. A: halogen donors (donor and halogen atom); B: halogen acceptors plus a neighboring "any" atom to define the secondary angle.

Implementation. We transfer the acceptor arrays once and stream donor/halogen chunks. The kernel first applies the distance prefilter, then both angle windows using cosine bounds (no `acos` unless both pass), this optimizes performance by avoiding expensive math operations. Pairs that satisfy all three constraints produce a valid distance.

Ionic interactions (cation-anion)

Role mapping A: cations in protein; B: anions in ligand.

Implementation. Straight distance check; simple and bandwidth-bound.

Ionic Interactions (cation-aromatic ring)

Role mapping A: cations in protein; B: aromatic rings in ligand (represented by centroid and normal vector).

Implementation. The kernel computes centroid distance and the angle to the ring normal; pairs failing either constraint are suppressed (sentinel). Distances are returned for positives.

π–stacking

Role mapping. A: aromatic rings in protein (5 or 6 membered), each represented by its centroid, plane normal, and ring vertices; B: aromatic rings in ligand (5 or 6 membered) with the same representation.

Implementation. The GPU computes, for every ring-ring pair: the centroid distance, plane–plane angle (via $|\cos|$ of normals), and two normal–centroid angles (again via $|\cos|$); it uses a prefilter to check whether the cosine value falls within at least one of the sandwich or T–shape value windows in cosine space before calling `sqrt/acos`. Back on the CPU, we evaluate the final windows and, for T–shape, run a 2D inside-polygon test by projecting the candidate centroid onto the target ring plane and using a ray-casting parity check across ring edges. Positive pairs produce *Pi Stacking* records with the two ring centroids.

Metal coordination

Role mapping. A: metal centers; B: chelating atoms.

Implementation. Bandwidth-bound distance kernel identical in structure to hydrophobic but using the tighter cutoff. Because the per-thread arithmetic is minimal, this kernel often does not benefit from offloading unless $|A| \cdot |B|$ is very large.

3.2.7 Host–Device Interface and Output

Post-processing and CSV emission. After each kernel completes and its result matrix is copied back, the CPU iterates the dense grid and emits one CSV line per positive pair.

Chapter 4

Results

This section presents the results obtained through screenshots, graphs, and other visual representations.

4.1 Tools to compare the implementations

To evaluate and compare the CPU and GPU implementations, both versions of the program can be executed on a predefined set of N protein-ligand pairs located in the `testing_samples` folder. Each run generates output in the form of CSV files, where each file contains detailed information about the execution time of the functions responsible for the molecular interactions identified for a specific pair.

Two scripts must be executed: one in the CPU branch and one in the GPU branch:

```
# In the CPU branch
./support/testing_profile_CPU.sh N

# In the GPU branch
./support/testing_profile_GPU.sh N
```

Note: To ensure a fair and consistent comparison between the CPU and GPU implementations, it is recommended to use the same value of `N` *in both executions. This guarantees that the same set of input pairs is analyzed by both versions of the program.*

Table 4.1: Example of profiling results showing execution time distribution per task

Range	Time (%)	Total Time (ns)	Instances
Total Program	33.9	412,196,794	1
Identify Interactions	32.3	393,265,926	1
Hydrophobic Interaction	32.2	391,586,015	1
Input	1.2	14,564,597	1
Identify Prot. Substructs	0.3	3,606,947	1
Hydrogen Bond	0.1	853,036	1
Ionic Interaction	0.1	774,715	1
Identify Lig. Substructs	0.1	747,394	1
Pi Stacking	0.0	45,855	1
Halogen Bond	0.0	1,498	1
Metal Coordination	0.0	1,179	1

Note: The original profiling data also includes average, median, minimum, maximum, and standard deviation for each task. Since every task is executed exactly once (Instances = 1), these values are either identical to the total time or zero in the case of standard deviation, and are thus omitted here for clarity.

To perform a comparison between the CPU and GPU implementations, two support scripts must be executed in sequence:

```
# Generate per-sample comparisons between CPU and GPU runs
python3 support/performance/generate_comparisons.py

# Analyze the aggregated results
python3 support/performance/analyze_comparison.py
```

The first script, `generate_comparisons.py`, processes the output generated by the CPU and GPU implementations for each protein-ligand pair in the `testing_samples` folder. It produces a CSV file for each pair, containing a side-by-side comparison of the molecular interactions identified by both versions.

Table 4.2: Comparison of execution times between CPU and GPU implementations

Range	CPU Time (ns)	GPU Time (ns)	Speedup (%)
Total Program	384,542,302	412,196,794	-7.19
Identify Interactions	373,922,567	393,265,926	-5.17
Hydrophobic Interaction	372,224,398	391,586,015	-5.20
Input	7,976,968	14,564,597	-82.58
Identify Prot. Substructs	2,188,132	3,606,947	-64.84
Hydrogen Bond	843,506	853,036	-1.13
Ionic Interaction	809,583	774,715	4.31
Identify Lig. Substructs	444,113	747,394	-68.29
Pi Stacking	39,484	45,855	-16.14
Halogen Bond	1,491	1,498	-0.47
Metal Coordination	973	1,179	-21.17

Note: The speedup is calculated as the relative performance improvement of the GPU version with respect to the CPU version. Negative values indicate that the GPU version was slower for that specific task.

The second script, `analyze_comparison.py`, processes all the comparison CSV files previously generated. It performs two main tasks:

1. It prints a summary of the performance differences between the CPU and GPU implementations for each function or interaction type.

2. It generates a bar chart (saved as a PNG image) that visually represents the percentage speedup or slowdown of the GPU version relative to the CPU version, using the `matplotlib` library.

4.2 The GPU performance tradeoff

4.2.1 Overview

GPU acceleration can deliver substantial speedups for highly parallel, compute-intensive kernels. However, the end-to-end runtime of a scientific workflow is governed not only by device throughput but also by fixed costs and CPU-side work. In our protein–ligand analysis pipeline, these factors dominate for small to moderate systems and in practice often nullify the theoretical advantage of the GPU.

4.2.2 Where GPU overheads come from

Kernel launch latency. Each device kernel incurs a fixed launch cost. For short kernels, this latency can be comparable to (or exceed) the compute itself.

Host-device data movement. Copying inputs/outputs between CPU and GPU memories incurs latency and limited bandwidth relative to on device memory. If performed frequently or on small batches, these transfers dominate.

4.2.3 When GPUs outperform CPUs

GPUs excel when:

- (i) the workload exposes massive independent parallelism,
- (ii) the arithmetic intensity (operations per byte moved) is high,
- (iii) non-GPU stages are negligible.

Typical winning cases include large batched computations where each element triggers substantial evaluation (for example many 3D distances/angles to compute), executed in a small number of sufficiently heavy kernels.

4.2.4 What we observed in our pipeline

Small systems. Testing the program on proteins with < 500 atoms and ligands with < 150 atoms, we observed that the GPU implementation is slower than the CPU version. In these cases, the GPU's fixed costs—data transfer and kernel launch overhead—are not offset by sufficient parallel work, allowing the CPU to complete the operations more quickly through direct execution.

Small systems are a common use case when the binding pocket has already been identified. Since such pockets are usually composed of a limited number of atoms, they are representative of focused interaction studies.

Figure 4.1 shows a binding pocket (green) composed of 467 atoms, with its ligand (43 atoms) positioned inside. Running both CPU and GPU versions of the program, we collected the following execution time results for the computational functions:

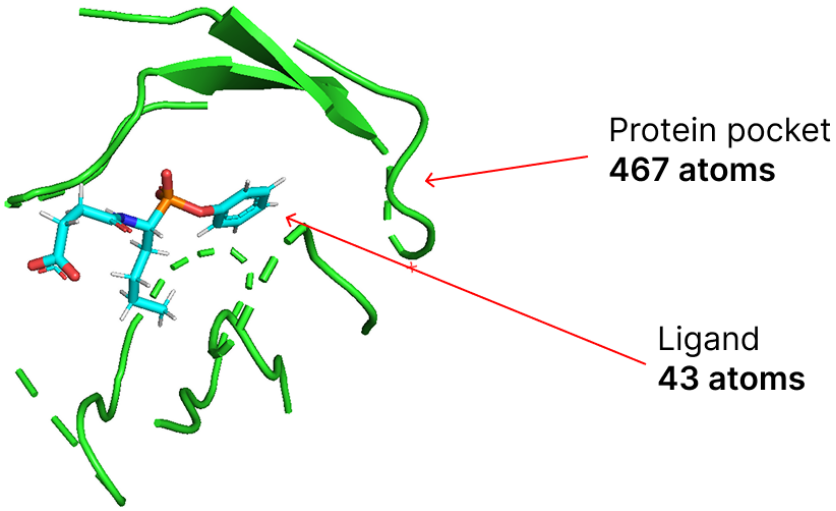

Figure 4.1: Example of a small binding pocket (green) with its ligand inside.

Table 4.3: Comparison of execution times between CPU and GPU implementations

Range	CPU Time (ns)	GPU Time (ns)	Speedup multiplier
Hydrophobic Interaction	381,135,342	400,620,368	-1.05
Hydrogen Bond	1,197,198	965,247	1.24
Ionic Interaction	1,263,149	854,984	1.48
Pi Stacking	49,805	33,126	1.50
Halogen Bond	1,751	1,143	1.53
Metal Coordination	1,338	2,288	-1.71

Table 4.4: Execution time comparison between CPU and GPU for a small system (467-atom pocket + 43-atom ligand).

As shown in Figure 4.2, the GPU program's overhead makes acceleration inefficient in this scenario, for the reasons discussed earlier.

Larger systems. As system size increases, the computational workload per operation grows, making better use of the GPU's parallel capabilities. Under these conditions, we observe that the GPU can outperform the CPU, as the larger workload amortizes the transfer and launch overheads.

Larger systems are relevant when the binding site is unknown and a more

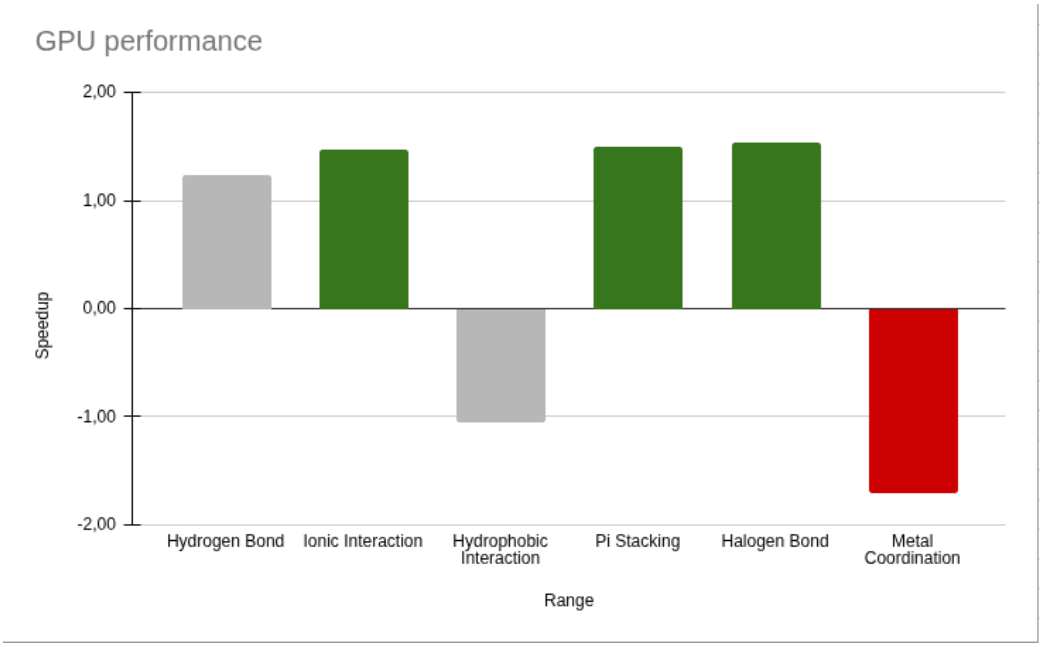

Figure 4.2: Performance comparison graph for the small system.

exhaustive analysis is required—such as scanning the entire protein surface for possible ligand interactions. In such cases, the higher atom counts provide enough parallel workload to exploit the GPU's architecture effectively.

Figures 4.3 and 4.4 show an example from PDB entry 6GTW (a crystallographic structure of the *Escherichia coli* ribosome) with 24,794 atoms, paired with ligand GCP (32 atoms) from the same PDB file.

In this case, we observe an overall increase in computational efficiency using the GPU. However, for three interaction types—ionic interactions, hydrophobic interactions, and metal coordination—the GPU shows no benefit or even worse performance. These three functions are the simplest in our set, each involving only a single distance calculation, so the amount of work per GPU thread is minimal.

Based on these observations, we tested a hybrid implementation, in which only the more computationally demanding interaction functions are parallelized, while the simplest ones remain on the CPU. The results of this hybrid approach are shown in Table 4.5.

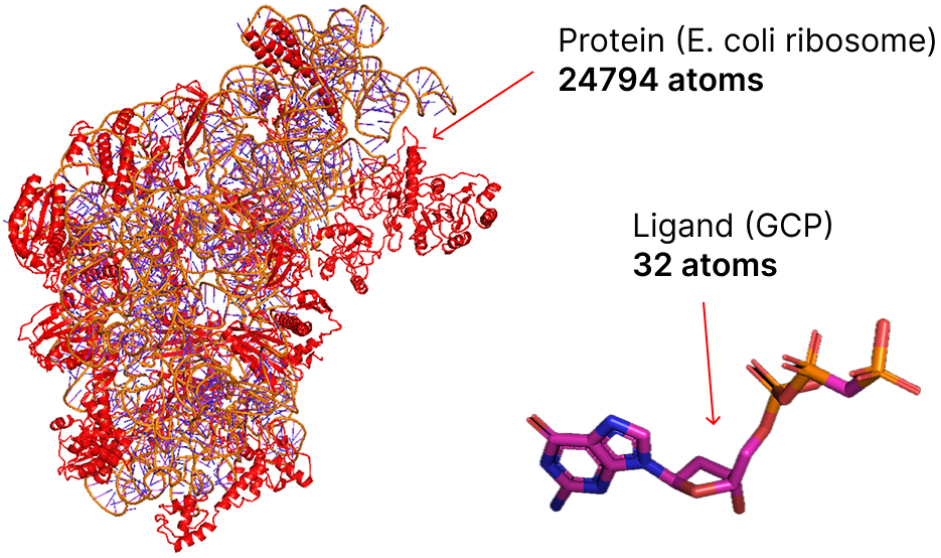

Figure 4.3: Visualization of the large protein structure from PDB 6GTW.

Table 4.5: Comparison of execution times between CPU and GPU implementations

Range	CPU Time (ns)	GPU Time (ns)	Speedup multiplier
Hydrophobic Interaction	441,559	434,609	1.02
Hydrogen Bond	4,703,659	820,946	5.73
Ionic Interaction	2497715	2,655,345	1.06
Pi Stacking	2,615	914	2.86
Halogen Bond	2,132	1,135	1.88
Metal Coordination	1,012	966	1.05

Table 4.6: Execution time comparison between CPU and GPU for a large system (24794-atom pocket + 32-atom ligand).

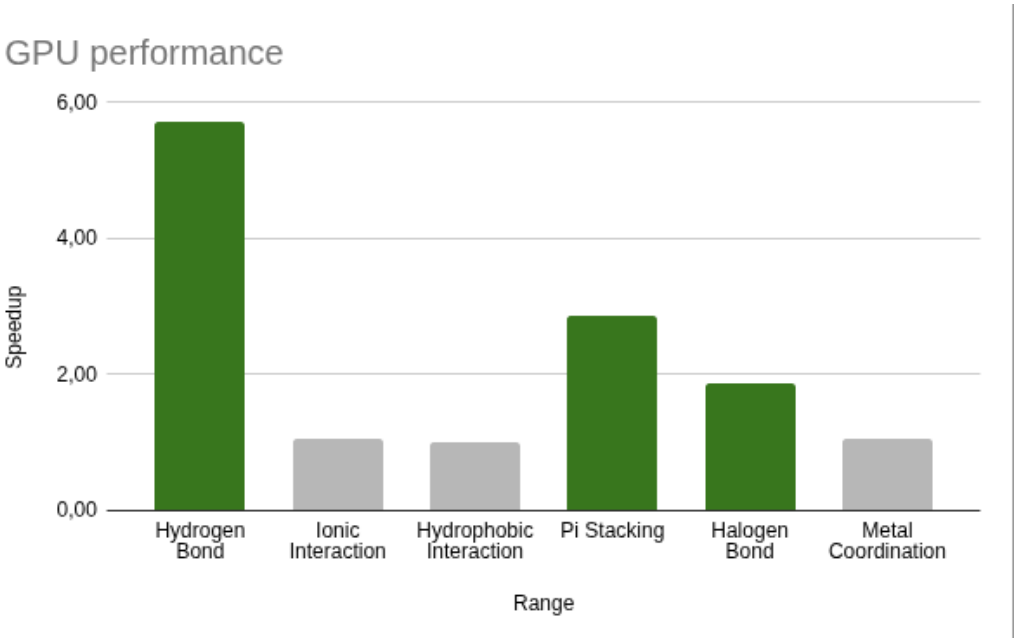

Figure 4.4: Visualization of ligand GCP from PDB 6GTW.

Chapter 5

Innovation of the Project

The primary innovation of this project lies in the hardware acceleration of a C++ code using CUDA technology. The code is designed to analyze interactions between a given protein and multiple ligands by utilizing the RDKit library, which is well-regarded in cheminformatics for its robust tools for molecular analysis.

5.1 Key Innovations

5.1.1 Enhanced Computational Efficiency

The project's most significant innovation is the use of CUDA to accelerate the computationally intensive tasks involved in protein-ligand interaction analysis by offloading the most demanding parts of the code to the GPU. This acceleration allows for faster processing of large datasets and more complex simulations, making the analysis more efficient.

5.1.2 Scalability

The hardware-accelerated solution is more scalable, meaning it can handle more extensive datasets and more complex interactions without a proportional increase in processing time. This scalability opens up new possibilities for large-scale studies that were previously limited by computational resources.

5.2 Impact of the Innovation

The acceleration of molecular interaction analysis through CUDA can have a wide-reaching impact across several fields. For instance, in drug discovery, the ability to rapidly analyze multiple protein-ligand interactions can accelerate the identification of potential drug candidates, thereby reducing the time and cost associated with the drug development process. In bioinformatics, this innovation allows for more complex and detailed modeling of molecular interactions, leading to better understanding and insights into biological processes.

Overall, the hardware acceleration introduced in this project represents a significant leap forward in computational efficiency, making it a valuable tool for researchers and professionals in various scientific domains.

Chapter 6

Project Setup Guide

6.1 Prerequisites

Before building and running the project, ensure the following tools are installed on your system:

- **GCC** — C++ compiler. Download GCC
- **NVCC** — CUDA C++ compiler (from CUDA Toolkit). Download CUDA Toolkit
- **RDKit** — Open-source cheminformatics toolkit. RDKit Installation Guide
- **CMake** — Build system generator. Download CMake

Note: It is recommended to use the latest stable versions.

6.2 Build Instructions

Use **CMake** to configure and build the project:

```
# Create a build directory
mkdir build

# Configure the project
cmake -S . -B build

# Build the project
cmake --build build
```

For additional information on CMake usage, refer to the Modern CMake Documentation.

6.3 Running the Program

After building the project, the program can be executed with the following command:

```
./build/interaction protein.pdb ligand1.mol2 ligand2.mol2
   ligand3.mol2 ...
```

The first argument must be the protein file in `PDB` format. Subsequent arguments are one or more ligand files in `Mol2` format.

There is no limit to the number of ligand files provided: the program will process all specified ligands after the protein.

6.4 Testing and Utility Scripts

The project provides two utility scripts located inside the `support/` directory:

- **Testing script** (`testing.sh`):
  - This script automatically applies the program to all sub-folders contained within the `testing_samples/` directory.
  - Each sub-folder must contain a protein file and one or more ligand files.
  - The output generated by the program is saved inside the corresponding sub-folder.
- **Delete script** (`deleteCSV.sh`):
  - This script removes all output files generated during testing from each sub-folder inside `testing_samples/`.

The scripts can be executed with the following commands:

```
# Run testing on all samples
./support/testing.sh

# Delete all generated output files
./support/deleteCSV.sh
```

6.5 How to generate the complete code documentation with Doxygen

The configuration file `Doxygen.txt` is already provided and correctly set up in the root directory of the project. To generate the documentation, follow these simple steps:

1. **Install Doxygen**:
   Make sure you have Doxygen installed. You can download it from `https://www.doxygen.nl/download.html` or install it via a package manager (e.g., `sudo apt install doxygen` on Debian-based Linux, `brew install doxygen` on macOS).

2. **Run Doxygen**:
   From the root folder of the project (the one containing `Doxygen.txt`), run the following command:

```
doxygen Doxygen.txt
```

3. **Visualize the output**:
   After the process completes, the documentation will be available inside the `doxygen_docs/html` directory. Open the `index.html` file in a web browser to view the documentation.

Chapter 7

Conclusions

In this project, the main achievement has been the development of a C++ program capable of analyzing protein-ligand interactions, built on top of the RDKit library. This required designing a pipeline to parse molecular structures and detect interaction patterns reliably across different input formats. Alongside this, an experimental CUDA implementation was also explored to assess the potential of GPU acceleration. While the GPU version has limitations in its current form, it provides a foundation for future optimization and highlights the possible benefits of hardware parallelism in more computationally demanding scenarios.

7.1 Future Work

While the current implementation has shown promising results, there are several avenues for future development:

- **Optimization**: Further optimization of the CUDA implementation could lead to even greater performance gains, particularly in the context of more complex molecular structures.
- **Expansion of RDKit Integration**: Expanding the integration with RDKit could include additional functionalities, such as more advanced cheminformatics algorithms or support for a broader range of molecular data formats.
- **Application Testing**: Applying the tool in real-world drug discovery or bioinformatics scenarios could validate its effectiveness and identify potential improvements based on practical use cases.

7.2 Final Thoughts

This project has been a valuable learning experience, allowing us to apply and deepen our knowledge in both software development and high-performance computing. As students of computer engineering, we have gained practical skills in C++ programming, CUDA optimization and molecular analysis, which are directly relevant to our field. Additionally, this project has reinforced our problem-solving abilities and our understanding of how to leverage advanced technologies to address complex scientific challenges. Overall, it has been a formative experience that has significantly contributed to our growth as future engineers.