TÉCNICO LISBOA

Force Field Calculation for Biomolecules

Pedro José Pereira Elias

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Dr. Francisco Jorge Dias Oliveira Fernandes
Prof. João António Madeiras Pereira

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes
Supervisor: Dr. Francisco Jorge Dias Oliveira Fernandes
Member of the Committee: Prof. Abel J.P. Gomes

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents and brother for their support, encouragement and caring over all these years and for always being there when i needed. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. João Madeiras Pereira and Prof. Francisco Fernandes for their constant help, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were a source of motivation during the good and difficult times. Thank you.

To each and every one of you – Thank you.

Abstract

Molecular Dynamics has been largely used in studies of biological ensembles such as proteins, amino acids, etc. With the increase of dimension and complexity of the problems to solve, it is crucial to constantly develop more optimized simulation algorithms to reduce computational costs and improve performance. To efficiently represent those high quantities of atoms of proteins, spatial acceleration data structures are required. In this work, it is proposed to perform a comparison analysis between data structures, both in CPU, including BVH, p-k-d trees and nested hierarchy, and in GPU, with two different BVH implementations. For the data structure comparison, an algorithm was used to calculate the neighbourhood of the atoms of the protein, mainly by using the distance between the atoms and an algorithm to traverse the data structure. This neighbourhood calculation allows a reduction in the number of pairwise interactions to be calculated in a molecular system, leading to a faster estimation of the total energy of the force field generated by the atoms.

Keywords

Molecular Dynamics, force fields, BVH, p-k-d trees, CPU, GPU

Resumo

A dinâmica molecular tem sido amplamente utilizada em estudos de agregados biológicos como proteínas, aminoácidos, etc. Com o aumento da dimensão e complexidade dos problemas a solucionar, é crucial o constante desenvolvimento de algoritmos de simulação mais otimizados para reduzir custos computacionais e melhorar o seu dempenho. Para representar eficientemente essas grandes quantidades de átomos é necessário usar estruturas de dados de aceleração espacial. Este trabalho propõe realizar uma análise comparativa entre estruturas de dados, tanto em CPU, incluindo BVH, árvores p-k-d e "nested hierarchy", e em GPU, com duas implementações de BVH diferentes. Para a comparação entre estruturas de dados, foi usado um algoritmo para calcular a vizinhança dos átomos da proteína, que usa principalmente a distância entre átomos, e um algoritmo que permite percorrer a estrutura de dados. O cálculo da vizinhança permite uma redução no número de pares de interações a serem calculadas num sistema molecular, o que leva a uma estimação mais rápida da energia total do campo de forças gerado pelos átomos.

Palavras Chave

Dinâmica molecular; campos de forças, BVH, árvores p-k-d, CPU, GPU

Contents

1 Introduction **1**
1.1 Objectives 3
2 Related work **5**
2.1 Molecular Dynamics 7
2.2 Force Fields 8
2.3 Simulation algorithms 10
2.3.1 Particle Mesh Ewald (PME) 10
2.3.2 Multilevel Summation Method (MSM) 11
2.4 Spatial Acceleration Data Structures 12
2.4.1 Grid 12
2.4.2 Cell Lists 12
2.4.3 K-d Tree 13
2.4.4 P-k-d Tree 14
2.4.5 Octree 15
2.4.6 Bounding Volume Hierarchy (BVH) 16
2.4.6.1 BVH construction 16
2.4.6.2 Applications of BVHs to Ray tracing 17
2.4.6.3 BVH improvements 17
2.4.7 Nested hierarchy 19
2.5 MD Simulation Tools 19
2.5.1 HOOMD-blue 20
2.5.2 LAMMPS 20
2.5.3 NAMD 21
2.5.4 AMBER 21
2.5.5 GROMACS 22
2.5.6 OPENMM 22
2.5.7 ACEMD 23

2.5.8 Overview . . . 23

3 Implementation 25

3.1 Architecture . . . 27

3.2 PDB file parser . . . 28

3.3 3D scene creation . . . 28

3.4 Spatial acceleration data structures . . . 28

3.4.1 BVH . . . 28

3.4.1.1 BVH CPU . . . 28

3.4.1.2 BVH GPU . . . 29

3.4.1.3 BVH8 GPU . . . 29

3.4.2 P-k-d trees . . . 30

3.4.3 Nested hierarchy . . . 30

3.5 OpenMP parallel implementation . . . 31

3.6 Visualization of the protein . . . 31

3.6.1 Visualization of the bounding boxes of the nodes of the spatial acceleration structures 32

3.7 Neighbourhood calculation algorithm . . . 34

3.8 CUDA implementation . . . 34

4 Tests and results 37

4.1 Ray tracing tests . . . 39

4.2 Neighbourhood tests . . . 41

5 Conclusion 47

5.1 Conclusions . . . 49

5.2 System Limitations and Future Work . . . 49

Bibliography 51

A Results Tables 57

List of Figures

2.1 Verlet algorithm pseudo-code [1] 8
2.2 Visual representation of the different potential energy terms in a force field [2] 8
2.3 Verlet list, representing the potential cut-off (solid circle) and the list range (dashed circle) [1]. 10
2.4 Steps of the MSM algorithm [3]. 11
2.5 3D representation of a Grid. 12
2.6 Standard cell list. The cells that each particle must search is illustrated in the shaded areas. 13
2.7 Stenciled cell list. The cells included in the stencil are indicated by the solid outlines of shaded areas. The area represented in a dashed line marks the cells in that stencil that are within the radius. 13
2.8 Partitioning of objects using k-d tree in 2 dimensions and the resulting binary tree. [4] . . 14
2.9 3D representation of an Octree with 2 levels of depth. [5] 15
2.10 Tree representation of the nodes of an Octree. [5] 15
2.11 2D representation of Bounding Volume Hierarchy. (a) Nearby particles are grouped into successively larger bounding boxes. (b) Hierarchical view of the bounding volume structure in (a). [6] 16
2.12 Example of a 2D Morton code ordering with the first two levels of the hierarchy [7]. 18
2.13 Illustration of a nested hierarchy structure, where each leaf node is a subset of the dataset. The leaf nodes are using p-k-d trees as a data structure and the parents of the leaf node are using BVH. 19
2.14 Neighbour list calculation based on a Hilbert curve. 22

3.1 Collapsing of a binary tree into the BVH8. 30
3.2 Visualization of a protein with 100 atoms. 32
3.3 Visualization of a protein with 100 atoms, including the bounding boxes of nodes using BVH CPU. 33
3.4 Visualization of a protein with 100 atoms, including the bounding boxes of nodes using p-k-d trees. 33

3.5 Visualization of a protein with 100 atoms, including the bounding boxes of nodes using NH. 34
3.6 Visualization of a protein with 100 atoms, where each atom is represented with a sphere, using the GPU path-tracer. 36
3.7 Visualization of a protein with 100 atoms, where each atom is represented with a triangle, using the GPU path-tracer. 36

4.1 Average FPS per number of threads. 39
4.2 Comparison of FPS between a single threaded implementation and the paralleled version. 40
4.3 Comparison of FPS between data structures in the rendering 40
4.4 Time required to find neighbours for the number of atoms per leaf node. 41
4.5 Memory occupied for the number of atoms per leaf node. 41
4.6 Time required to find neighbours for different NH thresholds. 42
4.7 Memory occupied for different NH thresholds. 42
4.8 Time required to find the neighbours of all atoms of the scene for sphere radius using the BVH GPU. 43
4.9 Time required to find the neighbours of all atoms of the scene in miliseconds (ms) by each acceleration structure. 44
4.10 Comparison of performance between each acceleration structure and CPU BVH. 44
4.11 Memory required to find the neighbours of all atoms of the scene by each acceleration structure. TH corresponds to the nested hierarchy threshold. 45
4.12 Comparison of efficiency between each acceleration structure and CPU BHV. TH corresponds to the nested hierarchy threshold. 45

List of Tables

2.1 Applications of BVHs. 17
2.2 Molecular Dynamics simulations tools. 20
2.3 Molecular Dynamics simulations tools. 24

3.1 BVH CPU structure. 29
3.2 BVH GPU structure. 29
3.3 BVH8 GPU structure. 30
3.4 PKD structure. 30
3.5 NH structure. 31

A.1 Time required to find the neighbours of 1000 objects in milliseconds (ms) by each acceleration structure. 57
A.2 Comparison of performance between each acceleration structure and CPU BVH. 58
A.3 Memory required to find the neighbours of all atoms of the scene by each acceleration structure. N is the number of atoms of the scene and TH corresponds to the nested hierarchy threshold. 58
A.4 Comparison of efficiency between each acceleration structure and CPU BHV. N is the number of atoms of the scene and TH corresponds to the nested hierarchy threshold. . . . 58

Acronyms

BVH Bounding Volume Hierarchy

k-d trees k-dimensional tree

p-k-d tree particle k-dimensional tree

NH Nested hierarchy

CPU Central Process Unit

GPU Graphics Processing Unit

PDB Protein Data Bank

PME Particle Mesh Ewald

MSM Multilevel Summation Method

CUDA Compute Unified Device Architecture

FPS frames per second

SVO sparse voxel octree

1 Introduction

Contents

1.1 Objectives 3

Proteins have a very important role in human life. Proteins like hemoglobin perform the transportation of oxygen through our body, and immunoglobulins, also known as antibodies, help us neutralise foreign entities such as viruses. All proteins are made of amino acids, and by combining amino acids in different ways, it is possible to obtain different proteins. Each amino acid is composed of atoms, and these atoms generate forces of attraction and repulsion between each other that behave like a force field and lead to a stable state of the protein. Researchers need to find out more about those interactions and about the composition of those proteins, but this is an expensive process to be done in a laboratory. By using simulations of particles in a 3D space that can be applied in many fields, from molecular dynamics to astrophysics, it is possible to calculate the interactions between atoms in a more affordable way. The data regarding the constitution of the biomolecules necessary to build the simulations can be obtained through the Protein Data Bank (PDB) [8]. PDB is a database that supports scientific research and education by providing free access to information about three-dimensional structures of macromolecules, such as proteins and nucleic acids. Molecular simulations are powerful tools commonly used in the biomolecular community that have become substantially more popular and visible in recent years. These simulations allow us to get information about the structure and dynamics of materials at the atomic level and to find promising structures or properties. However, these simulations can involve millions of particles, requiring many calculations and raising the need to speed up the process to make them faster but at the same time without using too much memory. To reduce this computational cost, the data is usually stored in a spatial data structure like grids, Bounding Volume Hierarchies (BVHs), octrees, k-dimensional tree (k-d trees), or a hybrid of them. It is also possible to reduce the time it takes to calculate the forces between particles, by using an efficient simulation algorithm like Particle Mesh Ewald (PME) or Multilevel Summation Method (MSM).

1.1 Objectives

The objective of this work is to use different acceleration structures, including BVHs, particle k-dimensional trees (p-k-d trees) and the Nested hierarchy (NH) that will be described later, which is a hybrid between the last two, that can be used to accelerate the heavy computation of the interactions of a force field that contains many atoms that compose a protein. More specifically, the goal is to accelerate the process of the calculation of the neighbourhood of atoms based on distances between each of those atoms and an algorithm that allows efficient traverse of the data structure.

2

Related work

Contents

2.1 Molecular Dynamics 7
2.2 Force Fields 8
2.3 Simulation algorithms 10
2.4 Spatial Acceleration Data Structures 12
2.5 MD Simulation Tools 19

In this section, more details on Molecular Dynamics (2.1), are presented, including the methods to calculate the energy components associated with the force field (2.2) by using the distance between particles. Also, it is explained how each of the simulation algorithms (2.3) and the data structures (2.4) work to speed up the calculation of the distance between atoms. Finally, there is a description of some of the simulation tools (2.5) including the improvements over the previous simulation algorithms, and some overview tables are presented which contain more detailed information about the studied approaches.

2.1 Molecular Dynamics

The molecular dynamics' classical approach algorithm [9] usually uses Newton's equations of motion. Considering a system with N particles, where m_i is the mass of the particle with index i, and f_i the force acting on it:

$$m_i \ddot{\overrightarrow{r_i}} = -\frac{\partial U_{pot}}{\partial \overrightarrow{r_i}} = \overrightarrow{f_i} \tag{2.1}$$

The potential is U_{pot}, where r_{ij} is the distance between the particles i and j:

$$U_{pot} = \sum_{i=1}^{N-1} \sum_{j>i}^{N} U\left(\overrightarrow{r_{ij}}\right), \overrightarrow{r_{ij}} = \overrightarrow{r_i} - \overrightarrow{r_j} \tag{2.2}$$

Furthermore, the energy, or hamiltonian, can be expressed as a sum of kinetic and potential terms:

$$E = E_{kin} + U_{pot} = \sum_{i=1}^{N} \frac{1}{2} m_i \dot{\overrightarrow{r_i}}^2 + U_{pot} \tag{2.3}$$

Since the calculation of these forces is expensive, the objective is to perform them as infrequently as possible. To ensure it is fast enough, it must increase the timestep without jeopardising energy conservation. One solution would be to use the Verlet algorithm [1], which may be may be expressed by the following equations:

$$p_i\left(t + \frac{1}{2}\delta t\right) = p_i\left(t\right) + \frac{1}{2}\delta t f_i\left(t\right) \tag{2.4}$$

$$r_i\left(t + \delta t\right) = r_i\left(t\right) + \delta t p_i\left(t + \frac{1}{2}\delta t\right)/m_i \tag{2.5}$$

$$p_i\left(t + \delta t\right) = p_i\left(t + \frac{1}{2}\delta t\right) + \frac{1}{2}\delta t f_i\left(t + \delta t\right) \tag{2.6}$$

The Verlet algorithm is time reversible, permits long timesteps, and just requires one force evaluation per step. The procedure is illustrated in the pseudo-code in fig 2.1.

```
do step = 1, nstep
   p = p + 0.5*dt*f
   r = r + dt*p/m
   f = force(r)
   p = p + 0.5*dt*f
enddo
```

Figure 2.1: Verlet algorithm pseudo-code [1]

2.2 Force Fields

In computer simulations, it is usually adopted the practice of not representing intramolecular bonds as terms in the potential energy function, and instead assume that the bonds are constrained to a fixed length. To calculate the energy of the system of a force field, by using the Born–Oppenheimer approximation, it is possible to separate the nuclear and electron cloud motions of the molecules and then discard the electronic part and only consider the nuclear coordinates of the particles. This way, in the calculation of the distance between particles, a particle can be represented by its position only.

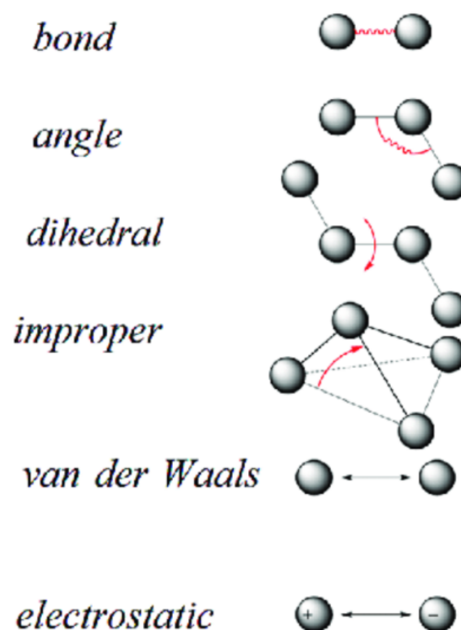

Figure 2.2: Visual representation of the different potential energy terms in a force field [2]

For energy calculation, most classical force fields [10] consider the potential energy terms associated with the sum of the following terms:

- deformation of bonds 2.7:

$$\sum_{bounds} \frac{k_d}{2}(d - d_0)^2 \tag{2.7}$$

- angle geometry 2.8

$$\sum_{angles} \frac{k_\Theta}{2}(\Theta - \Theta_0)^2 \tag{2.8}$$

- rotation about certain dihedral angles 2.9 and 2.10

$$\sum_{dihedrals} \frac{k_\Phi}{2}(1 + cos(n\Phi - \Phi_0)) \tag{2.9}$$

$$\sum_{impropers} \frac{k_\Psi}{2}(\Psi - \Psi_0)^2 \tag{2.10}$$

- van der Waals forces 2.11

$$\sum_{non-bounded\ pairs(i,j)} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \tag{2.11}$$

- electrostatic interactions 2.12

$$\sum_{non-bounded\ pairs(i,j)} \frac{q_i q_j}{4\Pi\varepsilon_0 r_{ij}} \tag{2.12}$$

For simplification, some works only consider the deformation of the bonds (eq. 2.7) and the non-bonded terms, van der Waals forces (eq. 2.11) and electrostatic interactions (eq. 2.12) for the energy calculation.

In eq. 2.7, k_d represents the constant force of the bond, d_0 represents the reference bond length, that is the equilibrium length when all other terms in the potential energy function are zero.

The van der Waals forces (eq. 2.11), are also known as Lennard-Jones potential (LJ), where r_{ij} is the distance between the particles i and j, ϵ is the depth of the potential well, and σ is the distance at which the particle-particle potential energy is zero. It is used between long distance particles according to the van der Waals component of the potential. This expression describes both attractive and repulsive forces.

The Coulombic electrostatic potential (eq. 2.12) is calculated based on the atomic nuclear charges.

In the computation of the non-bonded contribution of the forces in a simulation, there is a large number of pairwise calculations, as for each atom it is calculated the distance to each one of the other atoms. A way to improve the results is by using neighbour lists. Those lists store all the nearby atoms of a particular atom. One example that uses this approach is the Verlet list [1] that defines a sphere radius r_{cut} around an atom to represent the potential cut-off and another larger sphere radius r_{list} as

shown in the fig 2.3. The elements within the cut-off range are the ones considered, corresponding to the neighbours that are in the interaction range. The list that contains the elements inside the sphere with radius r_{list} needs to be constantly updated to avoid that unlisted pairs get in the interaction range.

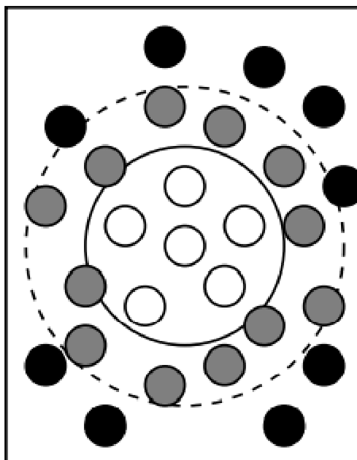

Figure 2.3: Verlet list, representing the potential cut-off (solid circle) and the list range (dashed circle) [1].

Some similar force field implementations like Amber [11], CHARMM [12], GROMOS [13], and OPLS-AA [14] have been proposed, although there are some differences. In the bonded components, the CHARMM and GROMOS force fields have the term for improper dihedral energy (eq. 2.10), but implementations such as Amber and OPLS-AA do not have a separate term. CHARMM also provides an additional angle term that handles the two terminal atoms in an angle, by adding a quadratic term that depends on the distance between the atoms. Regarding the non-bonded interactions, in the determination of the LJ parameters ε_{ij} and σ_{ij}, OPLS-AA and GROMOS force fields use geometric combining rules for both parameters, while CHARMM and Amber use the geometric mean for calculating ε_{ij}, but use the arithmetic mean for σ_{ij}.

2.3 Simulation algorithms

In this section, it is explained in more detail how each of the simulation algorithms work to make the computation more efficient, including the Particle Mesh Ewald and the Multilevel Summation Method.

2.3.1 Particle Mesh Ewald (PME)

PME [15] is a simulation method used to speed up the computation of long-range electrostatic forces in molecular dynamics simulations. To calculate those forces, there are two main contributions:

- Direct sum and neighbour list:

- The atoms are sorted spatially into boxes using the Hilbert curve, which is a continuous curve that fills the space and allows to improve space locality.

- Reciprocal sum:

  - Charge spreading onto a charge grid Q

  - 3D Fast Fourier Transform (FFT) [16] of Q from real to complex

  - Energy computation in reciprocal space

  - 3D Fast Fourier Transform (FFT) of the convolution from complex to real

  - Force computation per atom in real space

The Smooth Particle Mesh Ewald (SPME) [17] improves the efficiency of the algorithm by making an approximation of the long-range forces.

2.3.2 Multilevel Summation Method (MSM)

The multilevel summation method [3] offers an efficient algorithm to calculate long-range forces in molecular dynamics simulations.

MSM approximates the Coulombic potential energy by separating the interactions into short-range and long-range. The short-range part evaluates all particle pair interactions within a determined distance. The long-range becomes more efficient by using different grid spacings in the interpolation when calculating the grid point intermediate charge and potential.

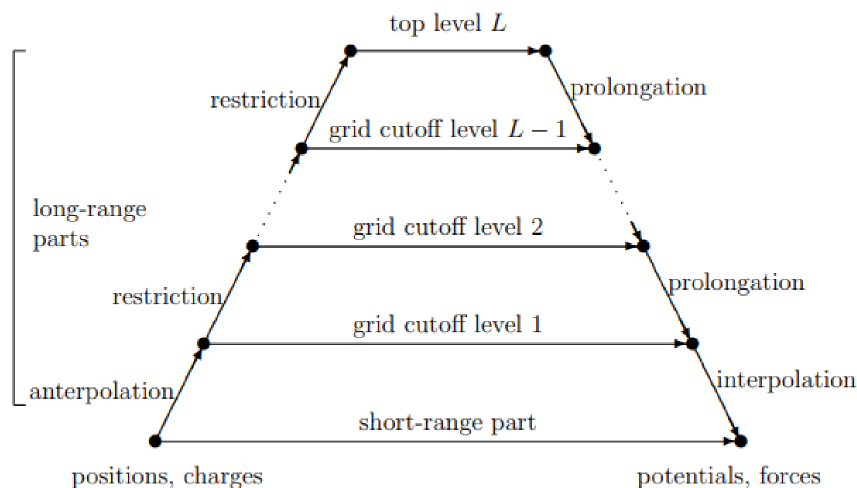

Figure 2.4: Steps of the MSM algorithm [3].

In fig 2.4, the gridded charges are calculated for each level moving up, and the gridded potentials are calculated moving down. The horizontal arrows represent the calculation of the short-range part and the grid cutoff calculations for each grid level.

2.4 Spatial Acceleration Data Structures

2.4.1 Grid

The idea of this spatial data structure is to subdivide the space into cells of the same size. The size of each cell for that specific dimension depends on the number of cells, N, and the size of the space, S, where the size of each cell = S / N. Grids are especially effective when the objects are uniformly spread through all the cells, in other words, when there are not empty cells, neither cells with too many objects. In simulations with too many particles that are not spread uniformly, this regular partitioning might have a large memory requirement. To solve this, a hash function is usually implemented to generate a 1D hashed table. If the implementation is efficient enough, it can reach a search time for finding an object of $O(1)$.

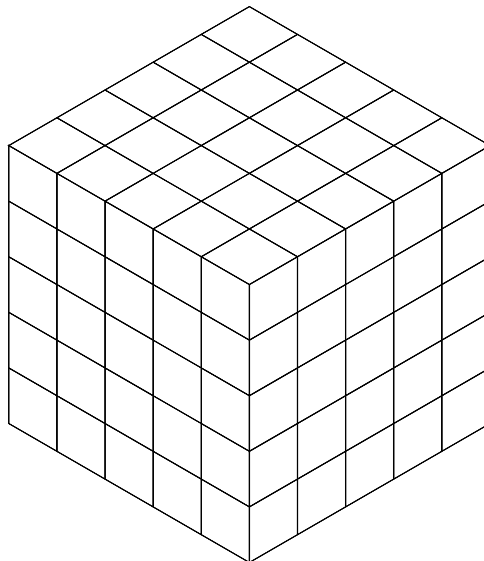

Figure 2.5: 3D representation of a Grid.

2.4.2 Cell Lists

A standard cell list consists of subdividing the space into uniform cells. The distance calculation only requires checking the adjacent cells, reducing the computation to O(Nm), where N is the number of particles and m is the average number of particles in a cell.

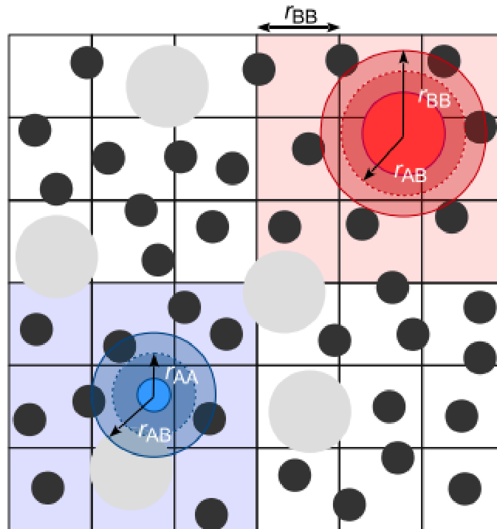

Figure 2.6: Standard cell list. The cells that each particle must search is illustrated in the shaded areas.

A stenciled cell list [18] is an extension of the standard cell list, where each type of particle has a maximum cutoff radius, and a stencil is computed based on the neighbouring cells.

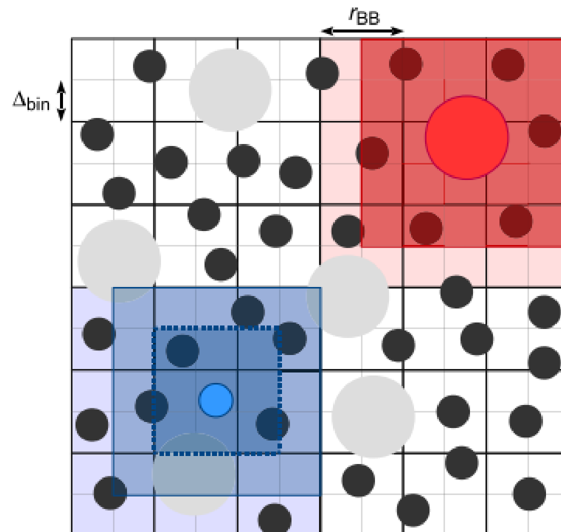

Figure 2.7: Stenciled cell list. The cells included in the stencil are indicated by the solid outlines of shaded areas. The area represented in a dashed line marks the cells in that stencil that are within the radius.

2.4.3 K-d Tree

A k-d tree is a space-partitioning data structure for organising points in a k-dimensional space. The k-d tree is a binary tree in which every non-leaf node generates an axis-aligned hyperplane that divides the space into two parts. It uses the medium cut recursively, often starting by the x-coordinate, then the

y-coordinate, then the z-coordinate, and then back to the x-coordinate and so on. The maximum depth is determined when the leaf nodes only contain a specific number of objects or when the threshold of the depth is reached. The easiest way to find the splitting plane is by sorting the objects and finding the median object. To find the neighbours of an object in a k-d tree implementation, it only requires O(logN) inspections on average, where N represents the number of nodes. This happens since the tree needs to be traversed until it reaches the leaf node.

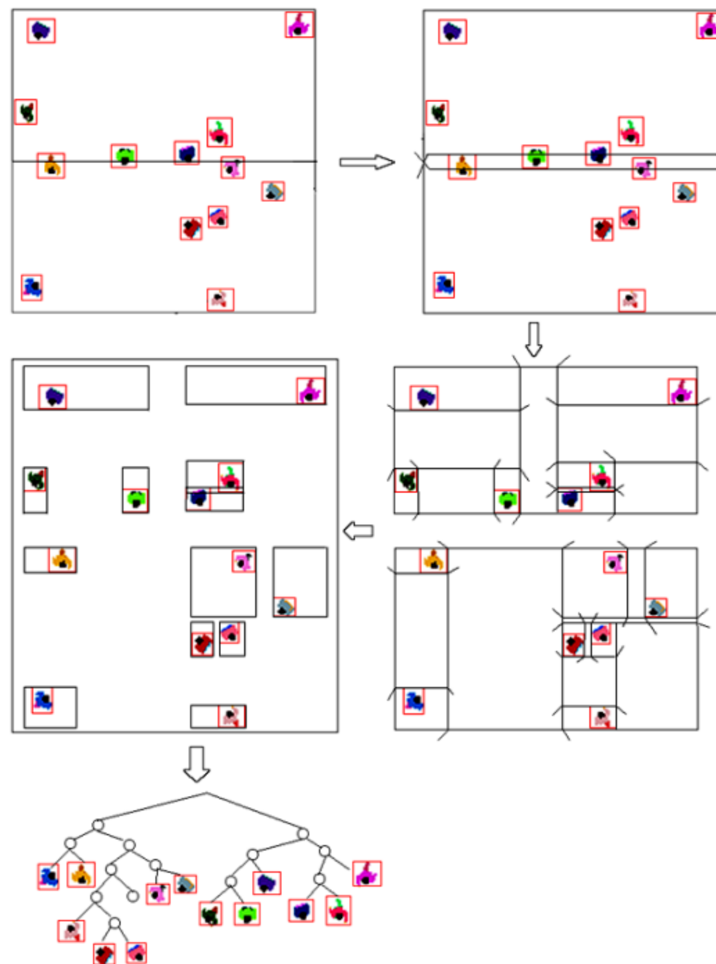

Figure 2.8: Partitioning of objects using k-d tree in 2 dimensions and the resulting binary tree. [4]

2.4.4 P-k-d Tree

A p-k-d tree [19] is a k-d tree adapted to particles where the dividing plane is encoded using the coordinate position of a particle. This means that no extra information is needed to store the splitting planes, and thus there is no memory overhead.

The construction algorithm generates a binary tree. This allows storing the information of the tree

in an array where the children of a node with index i have the indexes 2i + 1 and 2i + 2. In the first iteration of the algorithm, the pivot element that is chosen to separate the left and the right of the tree is the element that will be considered the root of the tree and will get the first position of the array. The algorithm continues recursively with the children of the pivot until the array is fully ordered.

2.4.5 Octree

An octree is a tree data structure that uses an axis-aligned hierarchical partitioning of a three-dimensional space. Each non-leaf/internal node in the octree has eight children, each child corresponding to an octant of the parent node. The main approach makes a new subdivision every time there is more than one point in a region. The octree generally uses less memory when compared to the grid in cases where there is a non-uniform distribution of the particles. However, it requires more search time when compared to the grid.

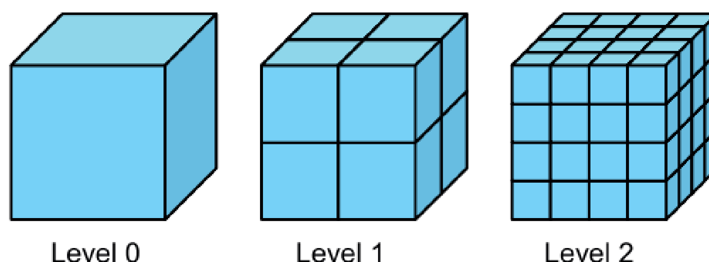

Figure 2.9: 3D representation of an Octree with 2 levels of depth. [5]

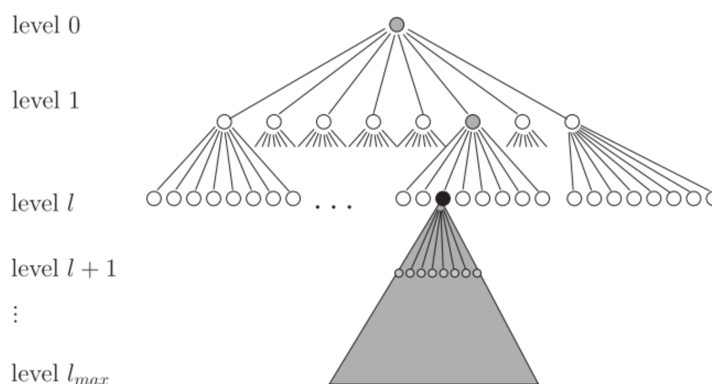

Figure 2.10: Tree representation of the nodes of an Octree. [5]

2.4.6 Bounding Volume Hierarchy (BVH)

Considering S as the set of N geometric objects, a bounding volume hierarchy (in this paper they call it a BV-tree) [20] is a tree data structure where each node, v, corresponds to a subset $Sv \subset S$, where the root node contains the whole set S. Each internal node (non-leaf node) contains two or more children. The number of nodes is at most 2N – 1.

In other words, all objects are wrapped in bounding volumes and each object is located in one of the leaf nodes of the tree. These leaf nodes are constantly getting agglomerated by combining two smaller nodes into a larger bounding volume until it reaches the top of the tree that contains all objects. This approach is faster than k-d trees and easier to implement, but has a higher memory cost.

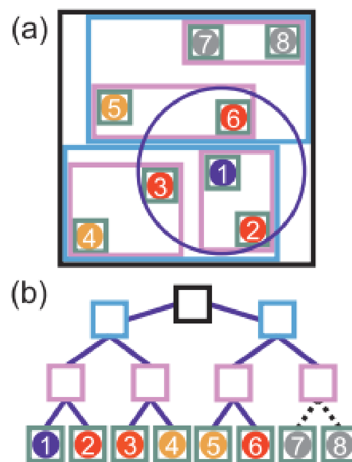

Figure 2.11: 2D representation of Bounding Volume Hierarchy. (a) Nearby particles are grouped into successively larger bounding boxes. (b) Hierarchical view of the bounding volume structure in (a). [6]

2.4.6.1 BVH construction

There are several ways to construct a BVH, including top-down and bottom-up approaches. The construction of a linear BVH [21] consists of the following steps:

- Construct AABBs: compute the bounding boxes for each of the objects.
- Calculate the scene bounding box that contains all objects.
- Assign Morton codes for each AABB.
- Sort the bounding boxes using Morton code.
- Generate the bounding volume hierarchy.
- Calculate internal nodes bounding boxes by traversing the tree bottom-up.

Morton codes, also known as Z-order codes, allow one to map data from multiple dimensions into a single dimension without losing spatial locality, as shown in fig 2.12.

The generation of the bounding volume is done in a recursive partitioning so that each internal node in the BVH tree corresponds to an interval of Morton codes. The recursion ends when it only contains one object. The partition is determined by a split that does the division of the interval on the place that it has the highest differing bits of Morton codes.

2.4.6.2 Applications of BVHs to Ray tracing

One common application of the bounding volume hierarchies is in ray tracing. In ray tracing, since it is necessary to test a lot of interceptions with all the objects to determine the colour of the pixel, it is required to have an acceleration structure to help it reduce the number of interceptions, making BVH one of the best options.

Table 2.1: Applications of BVHs.

Reference	Architecture		Ray Tracing	Improvements	
	CPU	GPU		Construction	Structure
[Lauterbach09]		•	•	•	
[Pantaleoni10]		•	•	•	
[Kopta12]	•	•	•		•
[Karras12]		•		•	
[Karras13]		•	•	•	
[Domingues15]		•	•		•
[Kim16]	•	•	•		•
[Vinkler16]		•	•		
[Vinkler17]		•	•	•	
[Hendrich17]	•		•	•	
[Pérard-Gayot17]		•	•		•
[Chitalu18]		•			•
[Gralka20]	•	•	•		•

2.4.6.3 BVH improvements

[Lauterbach09] [7] proposes a linear bounding volume hierarchy (LBVH) implementation that uses a high parallel algorithm where the particles are ordered along a Z-order curve. This allows the closer objects to be sorted near each other so that they are represented in a single fixed sequence with the length of the number of objects. The tree can be constructed by splitting intervals recursively to create new nodes, as it is illustrated in fig 2.12.

There are several implementations of BVH in GPU related to ray tracing, like [Karras13] [22] that proposed an improvement to the parallel construction of BVHs in [Karras12] [21] that focus on improving the quality of the existing BVH by looking to local neighbourhoods of nodes and restructuring the tree to achieve the optimal structure, as well as doing an improved parallel splitting of triangles prior to the construction of the tree.

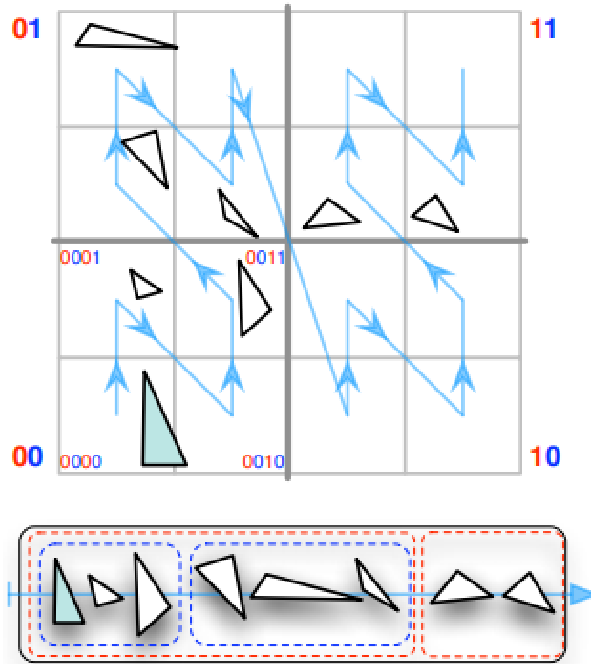

Figure 2.12: Example of a 2D Morton code ordering with the first two levels of the hierarchy [7].

Other study did a performance comparison between BVH and k-d trees [Vinkler16] [23] and an extended implementation of Morton Codes [Vinkler17] [24] that can encode the size of objects, has adaptive ordering of the codes, and can use variable bit counts for different dimensions.

[Pantaleoni10] [25] proposes a Hierarchical Linear Bounding Volume Hierarchy (HLBVH) that expands the LBVH [Lauterbach09] [7], by observing that the values assumed by the Morton code represent a voxel of a regular grid that includes the same objects of the corresponding bounding box and that higher bits represent the parent voxels as illustrated in 2.12. In the HLBVH implementation, it splits the original LBVH algorithm into a two-level hierarchy, where in the first level a sorting of the objects of the grid is performed only considering some of the higher bits and in the second level the remaining bits are used to do a sorting within each voxel.

In [Pérard-Gayot17] [26] it is used an approach with irregular grids, in [Kopta12] [27] the aim is to improve the efficiency of animated scenes and [Hendrich17] [28] proposes a new algorithm for constructing the BVH using CPU that uses a top-down approach with a progressively refined cut of an existing auxiliary BVH.

Some of these approaches, as well as recent GPU research, are beginning to use highly optimised ray tracing engines like OptiX to create interactive visualizations.

2.4.7 Nested hierarchy

In the work of [Gralka20] [29], it is proposed a ”nested hierarchy”, a hybrid acceleration data structure that is a composition of both BVH and p-k-d trees, where the nodes closer to the root use the BVH acceleration structure, but as we get closer to the leaf nodes, we switch to a p-k-d tree data structure.

This approach in terms of memory, when compared to BVH, requires about 10 times less of the size of the raw data during the build and then stabilises to about 3,5 times less of the raw data size. In terms of performance, the quantity of frames lost is not that significant when compared to BVH and has a significant gain in performance when compared to a p-k-d tree approach.

In terms of limitations, when applied to ray casting, it is known that it is only applicable to systems that are in some way limited by memory, otherwise using the BVH is a better choice since it has better performance.

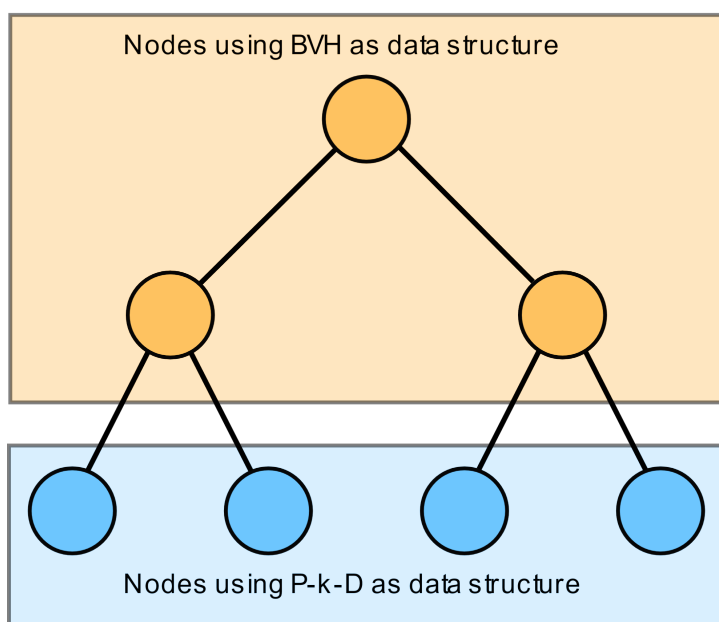

Figure 2.13: Illustration of a nested hierarchy structure, where each leaf node is a subset of the dataset. The leaf nodes are using p-k-d trees as a data structure and the parents of the leaf node are using BVH.

2.5 MD Simulation Tools

In this section, it is listed the main molecular dynamics simulation tools, such as HOOMD-blue, LAMMPS, NAMD, AMBER, GROMACS, OPENMM and ACEMD. In the following table, there is an overview of the tools, and in the subsections, there is more detailed information about each.

Table 2.2: Molecular Dynamics simulations tools.

Name	Source code (download link)	Programming Language	Documentation
HOOMD-blue	https://github.com/glotzerlab/hoomd-blue	Python, C++, Cuda	https://hoomd-blue.readthedocs.io/en/latest/
LAMMPS	https://github.com/lammps/lammps	C++	https://docs.lammps.org/Manual.html
NAMD	https://www.ks.uiuc.edu/Development/Download/download.cgi?PackageName=NAMD	C	https://www.ks.uiuc.edu/Research/namd/cvs/ug/
AMBER	https://github.com/Amber-MD	C++, Python	https://ambermd.org/Manuals.php
GROMACS	https://manual.gromacs.org/current/download.html	C++	https://manual.gromacs.org/current/index.html
OPENMM	https://github.com/openmm/openmm	C++, Python	http://docs.openmm.org/latest/userguide/

2.5.1 HOOMD-blue

In the work of [Anderson20] [30], they used the particle simulation engine called HOOMD-blue, a Python package for high-performance molecular dynamics. The current version already had many optimizations, including multiple threads per particle with warp-level reductions, atomic operations to build cell lists and many others. In this paper, based on [Anderson16] [31], it was added support for improved intra-node scaling to many GPUs and it was used a simulation algorithm called Hard Particle Monte Carlo (HPMC) [32] that consists of selecting a particle at random, generating a random move for that particle, checking for overlaps between the future configuration and all the other particles of the system, and then rejecting the move in case of no overlaps and accepting it otherwise.

HOOMD-blue has a BVH implementation that allows the creation of neighbour lists, which is two times faster than using the cell list.

They optimised all core functionalities so that it is possible to run these simulations as fast as possible, either on CPU or GPU depending on which one is best suited to the problem. All of this knowing that it is required to constantly refactor and rewrite to obtain the best performance on the latest hardware.

[Howard16] [33] presented efficient algorithms for molecular simulations for computing neighbour lists, one based on LBVHs and the other based on a stenciled cell list, and then compared the results with an implementation of a standard cell list.

The results showed that both the stenciled cell list and the LBVH algorithms outperform the standard cell list. They also demonstrated that LBVH has better performance when compared to the stenciled cell list.

2.5.2 LAMMPS

Previous implementations of particle–particle particle-mesh (P^3M) in LAMMPS consisted of three steps:

- charge assignment: each processor maps the charge on particles it owns to its 3D sub-section of the grid;

- field solve: performs 3d FFTs in parallel;
- interpolation: interpolation of electric field vectors to each of a processor's particles.

In the work of [Brown11] [34], algorithms for accelerating neighbour list builds and short-range force calculation were presented, while in [Brown12] [35] they focused on improving the first and third steps of the P^3M algorithm described above by improving the efficiency.

[Sirk13] [36] provides information about how to compute long-range electrostatic contributions. It proposes 2 different approaches:

- Extended Ewald summation method
- Extended particle-particle particle-mesh (P^3M) method

In the extended Ewald summation method, the potential energy calculation is based on the sum of two formulas, one similar to the eq. 2.12, and the other expressed with a Fourier series. The cost of this method is $O(N^{3/2})$. The P^3M does an interpolation of charges from atoms to a 3D grid. Then it applies a fast Fourier transformation (FFT) that makes the energy calculation possible. This method has cost of $O(Nlog(N))$ and scales better when compared to the Ewald summation.

2.5.3 NAMD

[Phillips20] [37] describes an implementation based on a Smooth Particle-Mesh Ewald [Essmann95] [17] that increases the speed by using an approximation on a grid to be used in the FFT. In the [Stone16] [38] implementation, all the PME computation moved to the GPU and changes were introduced in the way the charge spreading and force gathering were performed.

NAMD records some information from the PME energy calculation, which allows to speed up the calculation of expensive functions during the simulation. With this paper's approach, PME can calculate the spread of charge from atoms to grid points and from the grid forces to atoms. This GPU acceleration can yield better results than CPU versions by doubling the grid spacing and increasing the order of interpolation from 4 to 8, which does not impact the accuracy and reduces the communication bandwidth by a factor of 8. The complexity of Ewald sum for more distant atoms is $O(NlogN)$, instead of the $O(N^2)$ of a Naïve approach.

2.5.4 AMBER

In the work of [Salomon-Ferrer13] [16], there is a PME implementation that involves 2 main contributions:

- Direct sum and neighbour list, that combines 2 sorting algorithms:

- The atoms are sorted spatially into boxes. Each box is stored in the highest order bits of the sorting key.

- The other bits are used to encode the Hilbert curve to improve space locality.

• Reciprocal sum:

  - Charge spreading onto a charge grid Q

  - 3D Fast Fourier Transform (FFT) [16] of Q from real to complex

  - Energy computation in reciprocal space

  - 3D Fast Fourier Transform (FFT) of the convolution from complex to real

  - Force computation per atom in real space

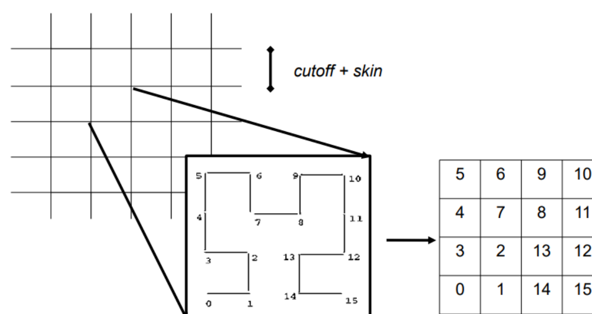

Figure 2.14: Neighbour list calculation based on a Hilbert curve.

2.5.5 GROMACS

In the work of [Páll15] [39], they developed a new approach where a fixed number of particles were grouped into spatial clusters. First, they placed the particles in a grid and then binned them in the z dimension. This allows to group the particles that are closer to each other in space, and to generate a list of clusters, each containing the same number of particles. A list was then constructed of all those cluster pairs containing particles that may be close enough to interact. In [Wennberg13] [40] and [Wennberg15] [41] some improvements regarding the Particle Mesh Ewald were provided, more specifically in the long-ranged Lennard-Jones interactions.

2.5.6 OPENMM

[Eastman10] [42] introduced the algorithm Constant Constraint Matrix Approximation (CCMA), based on the observation that the Jacobian matrix changes very little over the course of a simulation. That

reduces the computation by constraining the bond lengths in molecular simulations from a step size 1 fs, that is the case of standard molecular force fields with no constraints, to a step size up to 4 fs.

2.5.7 ACEMD

[Harvey09] provided an implementation of the Smooth Particle Mesh Ewald Method on GPU, which is similar to the reciprocal sum present in [Salomon-Ferrer13] [16] and added parallelism to distribute the work through threads.

2.5.8 Overview

Apart from the main references that have already been discussed, there are also other important references present in table 2.3, as [Lebrun-Grandie20] [43] that introduced a library that uses a BVH implementation to search for geometric objects that are close in a space or [Robinson17] [44] that includes a library for implementing particle-based methods such as neighbourhood searches and fast summation algorithms.

[Guntury12] [45] used the grid for ray tracing of scenes and the works of [Ogarko12] [46] and [Krijgsman14] [47] provide a fast implementation of contact detection in particle systems using multi-level grids, also called hierarchical grids.

[Bautembach11] [48] describes sparse voxel octrees (SVOs) and provides a solution to animate them whereas the studies of [Madoš20] [49] and [Madoš21] [50] suggest solutions to improve the efficiency of the SVOs.

Table 2.3: Molecular Dynamics simulations tools.

Reference	Name of the tool	Architecture		Acceleration structure				Simulation algorithm		
		CPU	GPU	List	Grid	SVO	BVH	PME	MSM	Other
[Harvey09]	ACEMD		•		•			•		
[Salomon-Ferrer13]	AMBER		•	Neighbour	•			•		
[Wennberg13]	GROMACS	•	•		•			LJ-PME		
[Páll14]	GROMACS		•		•			•		
[Wennberg15]	GROMACS	•	•		•			LJ-PME		
[Howard16]	HOOMD-blue		•	Stenciled			•			
[Anderson16]	HOOMD-blue		•	Cell			LBVH	•		HPMC
[Anderson20]	HOOMD-blue		•				LBVH			HPMC
[Brown11]	LAMMPS	•	•	Neighbour				•		
[Brown12]	LAMMPS	•	•	Neighbour	•			•		
[Sirk13]	LAMMPS	•			•			•		
[Hardy15]	NAMD	•	•		•				•	
[Stone16]	NAMD		•		•			•		
[Phillips20]	NAMD	•	•		•			•	•	
[Eastman10]	OPENMM	•	•							CCMA
[Robinson17]	Aboria	•			•					
[Lebrun-Grandie20]	ArborX	•	•				•			
[Bautembach11]		•				•				
[Guntury11]			•		•					
[Ogarko12]		•			Hierarchical					
[Krijgsman14]		•			Hierarchical					
[Madoš20]		•				•				
[Madoš21]		•				•				

3

Implementation

Contents

3.1 Architecture 27
3.2 PDB file parser 28
3.3 3D scene creation 28
3.4 Spatial acceleration data structures 28
3.5 OpenMP parallel implementation 31
3.6 Visualization of the protein 31
3.7 Neighbourhood calculation algorithm 34
3.8 CUDA implementation 34

3.1 Architecture

This work explores the use of different spatial acceleration data structures, more specifically the BVH, p-k-d tree and the nested hierarchy, to contain the information required to represent molecules, as well as organise them in a way facilitates computation. More specifically, the spatial data structures are applied to accelerate the computation of two different topics: (i) the visualization of the protein in a 3D scene that uses ray-tracing to render the scene and (ii) the calculation of the neighbourhood of the atoms, which is a crucial step in the calculation of the interactions between the atoms of the protein that simulate a force field.

This work is also divided into two implementations: one running in Central Process Unit (CPU) and the other one in Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA). CUDA consists of running a sequential host program that can initiate parallel kernels on the GPU device. Each kernel executes a single program across many threads that execute in parallel.

In the CPU implementation, the 3D rendering of the protein was performed using the information of a PDB file, based on a CPU ray-tracing implementation running in Open-GL using BVH as an acceleration structure. Then this solution was extended to allow other acceleration structures as p-k-d tree and NH. Later, to improve the performance of the calculations and increase the frame rate, an OpenMP parallel implementation was done to distribute the work by a number of CPU threads. For a better understanding of the data structures, it was also made a visualization of the bounding boxes of some nodes of the structure in use. Regarding the interactions between atoms, an algorithm was developed to find all neighbours of a specific atom by traversing the data structure.

In the GPU implementation, a GPU path-tracer implementation running in CUDA and using BVHs as acceleration data structures was used as a base. The objective was to not only visualize the protein in the scene but also to calculate the neighbourhood of the atoms and to analyze the performance when compared to the CPU.

In brief, the features implemented in this work are the following:

1. Parse the PDB file to gather the protein information.
2. Create a 3D scene with the parsed information.
3. Implement the acceleration structures.
4. Implement parallelism using OpenMP.
5. Visualize the protein using a ray-tracing render for each acceleration structure.
6. Visualize the bounding boxes of the nodes of the spatial acceleration structures of a specific object.
7. Create an algorithm to find the neighbours in the proximity of an atom for each data structure.

8. Implement in CUDA the steps 1, 5 and 7.

3.2 PDB file parser

Protein information is usually stored in PDB files with a ".pdb" extension. Those files contain many sections regarding different aspects of the protein, but the lines that are important for this work are the ones that start with "ATOM". Each of those records has the information of the sequential number, the atom name, the name and number of the residue it belongs to, the letter that specifies the chain, the x, y, and z coordinates, the occupancy, and the temperature factor. Although all of that information is useful for many studies, the most relevant for this work are the coordinates and the elements of the atoms. The coordinates are needed to map the atoms in 3D space, and the element of the atom is required for the visual representation of the protein.

3.3 3D scene creation

To have a visualization of the protein, it is required not only to parse the information from the PDB file but also to have 3D objects that are able to be rendered. Therefore, each atom was represented as a sphere with a center on the coordinates of the atom and a radius according to the respective element of the atom. Hence, a protein could be represented by a list of spheres.

3.4 Spatial acceleration data structures

In this section, it will be explained in more detail how each of the spatial acceleration data structures are implemented, both in the CPU, using the BVH CPU, the p-k-d tree and the nested hierarchy, and in the GPU, using the BVH GPU and the BVH8 GPU.

3.4.1 BVH

3.4.1.1 BVH CPU

The most relevant information in the class BVH is the list of the objects in the scene and the list of BVH nodes. Each BVH node contains a bounding box where all the objects of that specific node are in, a boolean to indicate if it is a leaf node or not, the number of objects it contains, the index to the left child if it is a non-leaf node, or the index to the next intersectable object in the objects vector if it is a leaf node, as shown in table 3.1.

Table 3.1: BVH CPU structure.

BVH	
objects	vector<Object*>
nodes	vector<BVH::BVHNode*>
threshold	int

BVHNode	
bounding_box	AABB
leaf	bool
n_objs	unsigned int
index_left_child	unsigned int
index_next_obj	unsigned int

In the construction of the BVH, the algorithm starts by determining the world bounding box that contains all objects in the scene. Then it builds recursively until it reaches the threshold that makes the recursion stop by having insufficient objects in each node. In each step of the recursion, it checks the dimension with greater width, orders the objects according to that dimension, and determines the "split index". The so-called "split index" is the index that splits the objects for each child node according to the mid point of the dimension with the higher width. It then creates the child nodes and the process repeats.

3.4.1.2 BVH GPU

As shown in the table 3.2, the information required to represent the structure is similar to the BVH CPU implementation. For the construction of the BVH, this approach uses a standard surface area heuristic (SAH) algorithm that is particularly efficient for ray-tracing, although it should not affect negatively the results in the neighbourhood tests since the construction phase time is not considered.

Table 3.2: BVH GPU structure.

BVH	
nodes	BVH2Node*

BVH2Node	
bounding_box	AABB
n_objs	unsigned int
left_child	int
index_next_obj	int

3.4.1.3 BVH8 GPU

The BVH8 GPU implementation is a compressed 8-wide BVH, which is constructed by collapsing a binary BVH, as represented in fig 3.1. This data structure stores a list of BVH8Nodes where each node contains 5 float4 elements that equals to 80 bytes per BVH8Node. This approach allows a reduction of memory traffic when applied for ray casting. This data structure works in a way that each parent node has 8 child nodes and all the required information to traverse it, including bounding boxes, indexes, etc, is encoded in the bits of the floats from node_0 to node_4 according to [51].

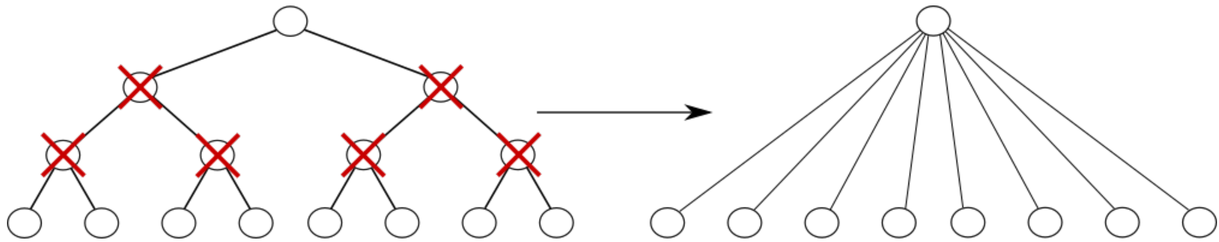

Figure 3.1: Collapsing of a binary tree into the BVH8.

Table 3.3: BVH8 GPU structure.

BVH8	
nodes	BVH8Node*

BVH8Node	
node_0	float4
node_1	float4
node_2	float4
node_3	float4
node_4	float4

3.4.2 P-k-d trees

The information contained in the PKD class is similar to the BVH class, the only addition being an integer "split index" that represents the index in the list of objects responsible for the split in child nodes, as represented in the table 3.4.

Table 3.4: PKD structure.

PKD	
objects	vector<Object*>
nodes	vector<PKD::PKDNode*>
threshold	int

PKDNode	
bounding_box	AABB
leaf	bool
n_objs	unsigned int
index_left_child	unsigned int
index_next_obj	unsigned int
split_index	int

The construction of the p-k-d tree has many aspects in common with the construction of the BVH. The parts where it is different are enumerated below. When selecting the dimension of the split, it is chosen first using x, then y, and then z, and so on. The other different aspect is that the object represented by the "split index" no longer belongs to the child nodes and will be exclusively represented in the parent node.

3.4.3 Nested hierarchy

The nested hierarchy class has the same information as the p-k-d tree class and has the advantage of being able to behave as both a BVH node and a PKD node depending on the number of objects it contains. It behaves as a BVH when the number of objects is higher than the NH_threshold established

and as a p-k-d tree if it is lower. As shown in the table 3.5, the variable NH_threshold is exclusive to this implementation, and no other information is required to be added to this implementation in order for its proper execution.

Table 3.5: NH structure.

NH	
objects	vector<Object*>
nodes	vector<NH::NHNode*>
threshold	int
NH_threshold	float

NHNode	
bounding_box	AABB
leaf	bool
n_objs	unsigned int
index_left_child	unsigned int
index_next_obj	unsigned int
split_index	int

The construction of the nested hierarchy is done by checking if it should be applied a BVH node or a p-k-d tree node to each of the nodes and then by reusing the parts of the code according to the data structure in use.

3.5 OpenMP parallel implementation

The objective of the OpenMP parallel implementation is to speed up the process of the ray tracer in order to reduce the time it requires to render each frame, which leads to an increase in the frame rate. This approach is based on the fact that for the rendering of a single frame, different threads may be used to divide the workload, more specifically, each thread can be responsible for a partition of the total number of pixels. The parallelism can be achieved by replacing the *for* loop that allows the program to access each pixel with a parallel *for* loop.

3.6 Visualization of the protein

In this section, it is shown the results after applying the ray-tracing algorithm to a scene that contains the information required to represent the protein. Independently of the data structure in use, the color of the pixels generated is the same since both the scene objects and the camera position are the same and the only goal of using different data structures is for the performance increase. Fig 3.2 show a protein representation according to the standard conventions, using a sphere with different radius values and colors based on the atom element where it is represented Hydrogen in white, Carbon in grey, Nitrogen in blue, Oxygen in red, Phosphorus in orange and Sulfur in yellow.

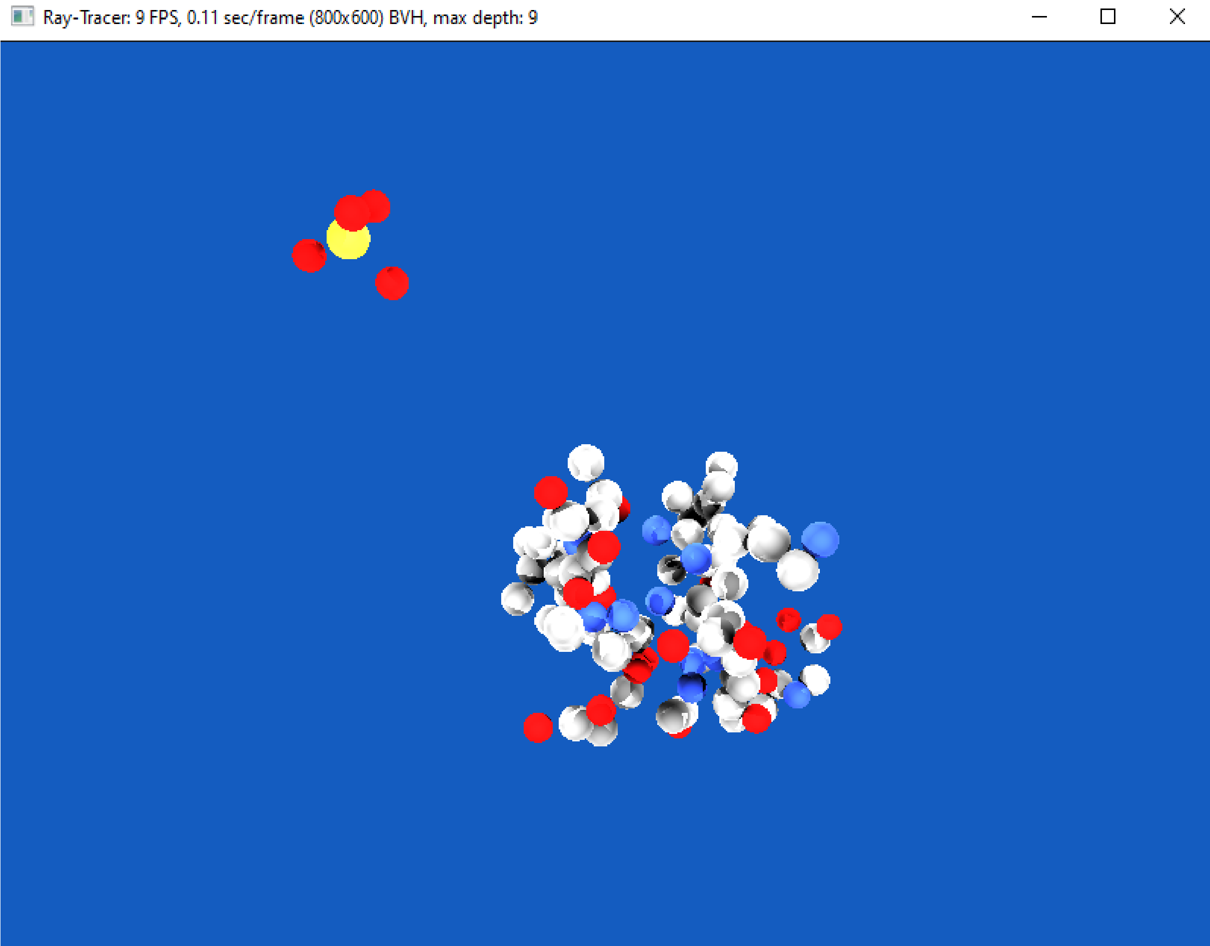

Figure 3.2: Visualization of a protein with 100 atoms.

3.6.1 Visualization of the bounding boxes of the nodes of the spatial acceleration structures

To visually understand more about the data structures, it was opted to implement the display of the bounding boxes of the nodes, more specifically the hierarchy of nodes that goes from the root node to the smallest node that contains a certain chosen atom. This was done by adding 12 objects to the scene for each bounding box to represent each of the edges and then rebuilding the data structure in use. For color choice, the BVH used a red-to-yellow gradient (fig 3.3), the p-k-d tree used a dark-to-light green gradient (fig 3.4), and the NH used the colors according to the data structure behaviour of each node (fig 3.5).

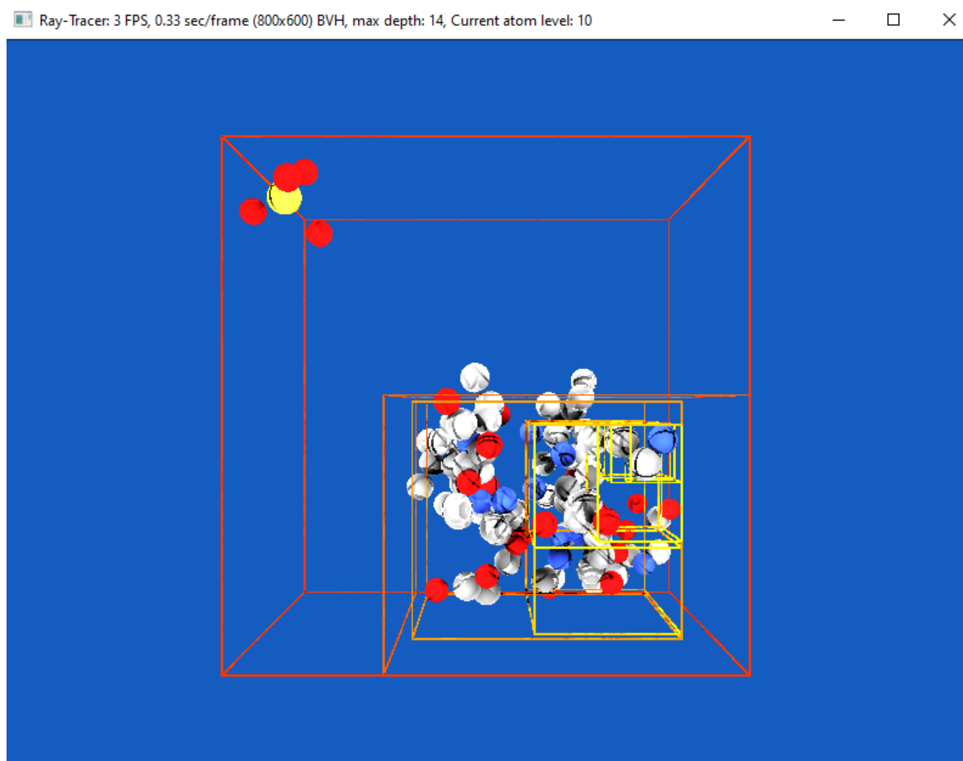

Figure 3.3: Visualization of a protein with 100 atoms, including the bounding boxes of nodes using BVH CPU.

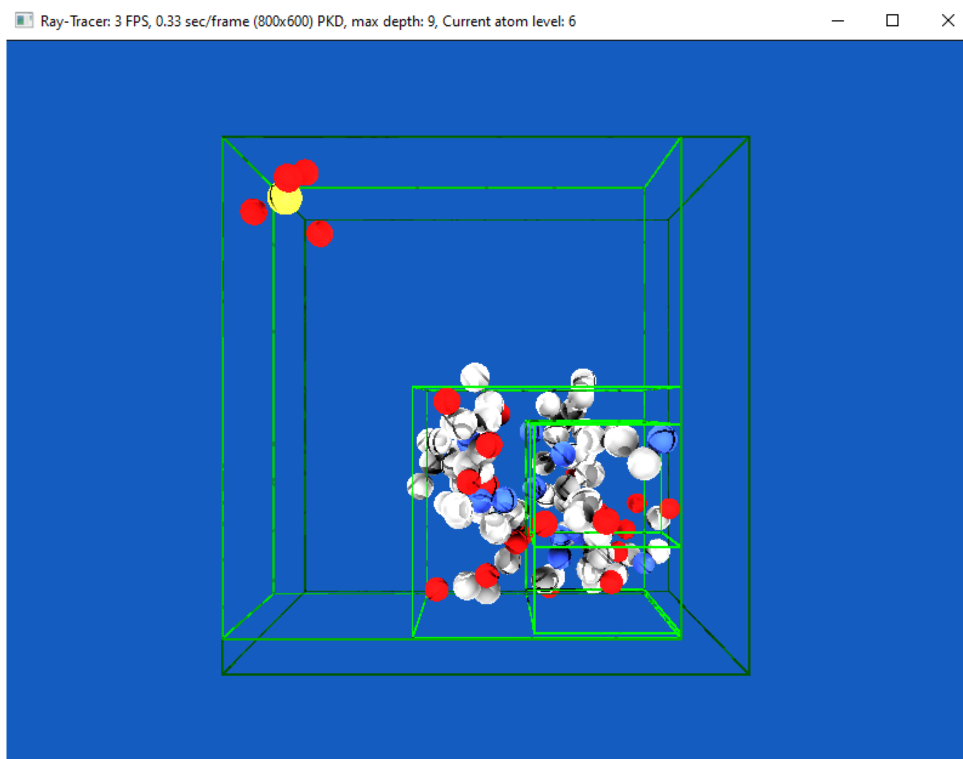

Figure 3.4: Visualization of a protein with 100 atoms, including the bounding boxes of nodes using p-k-d trees.

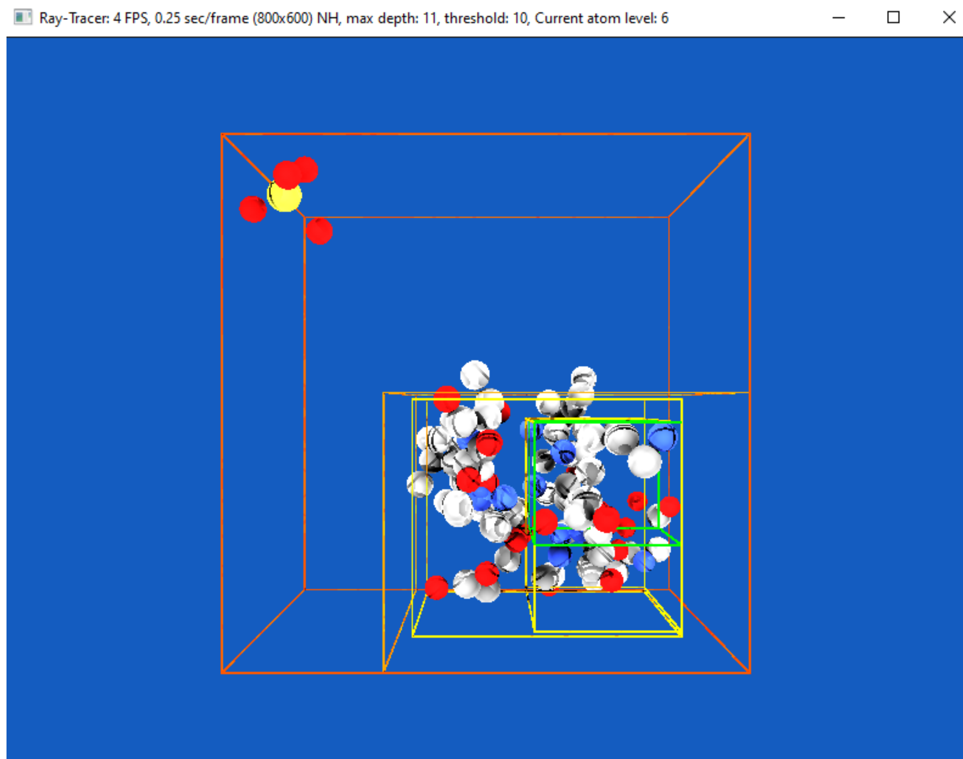

Figure 3.5: Visualization of a protein with 100 atoms, including the bounding boxes of nodes using NH.

3.7 Neighbourhood calculation algorithm

The neighbourhood calculation algorithm is responsible for the traversal of the spatial acceleration data structure in use and can be applied to both the BVHs, the p-k-d trees and the NH. The algorithm starts by picking an atom to calculate the neighbourhood that serves as the center of the proximity sphere. Then, starting from the root node, overlap checks of the sphere of proximity of the picked atom with the bounding box of that root node are performed. If it hits the root, it will do the same overlap check with each of the child nodes. For each child node intercepted, (i) if it is a leaf node, it checks if the atoms of that node are in the proximity of the sphere, adding the neighbours to a list; (ii) if it is a non-leaf node, it recursively checks its child nodes.

3.8 CUDA implementation

In this section, the development process regarding the CUDA implementation is described in more detail. The main approach was to find an implementation that uses BVH as data structure in GPU and then adapt it to do the neighbourhood tests similarly to the ones already implemented in CPU. It was discovered an implementation in CUDA that was using different variations of BVHs applied to

an interactive path-tracer algorithm [52] and it was decided to use this as the base code to the GPU implementation. Since there were many types of BVHs implemented, it was decided to work with only the most relevant ones, including a SAH-based BVH and a Compressed Wide BVH (BVH8) [51] that is constructed by collapsing a binary BVH, where each parent node contains up to 8 child nodes and each node takes up to 80 bytes.

At the start, the objective was to load the PDB file information to be able to initialize the data structure and create the scene. However, this implementation was using as input a file with the extention ".obj". To solve this problem, it was required to convert the PDB file to an ".obj" file. This was done by using the tool Blender, which was able to create a scene based on the PDB file and then export it as a ".obj". Also by default, Blender would load the PDB files as spheres for each atom, as shown in fig 3.6, which would generate many triangles in the ".obj" file, which was sufficient for the rendering of the scene but not convenient for making the neighbourhood tests. To solve this particular problem, a script was done to change the ".obj" that for each sphere would remove all triangles except the first triangle of that atom, which generated a scene like the fig 3.7. This allows to have an ".obj" file where each triangle represents an atom. However, this file conversion generated more problems, since the distance between atoms in the ".obj" file was changed and was no longer in the unit of Angstroms. By comparing the distances between the same atoms in each file, it was possible to obtain a correlation between the files and to estimate the distances in Angstroms. Since the goal was to not only visualize the protein but also do the neighbourhood calculation, this approach was adapted to, instead of considering an atom as a triangle, representing the atom as a coordinate in the space given by the first vertex of the triangle. Then, the traversal of each of the data structures was changed to do the neighbourhood calculation according to the algorithm in the section 3.7.

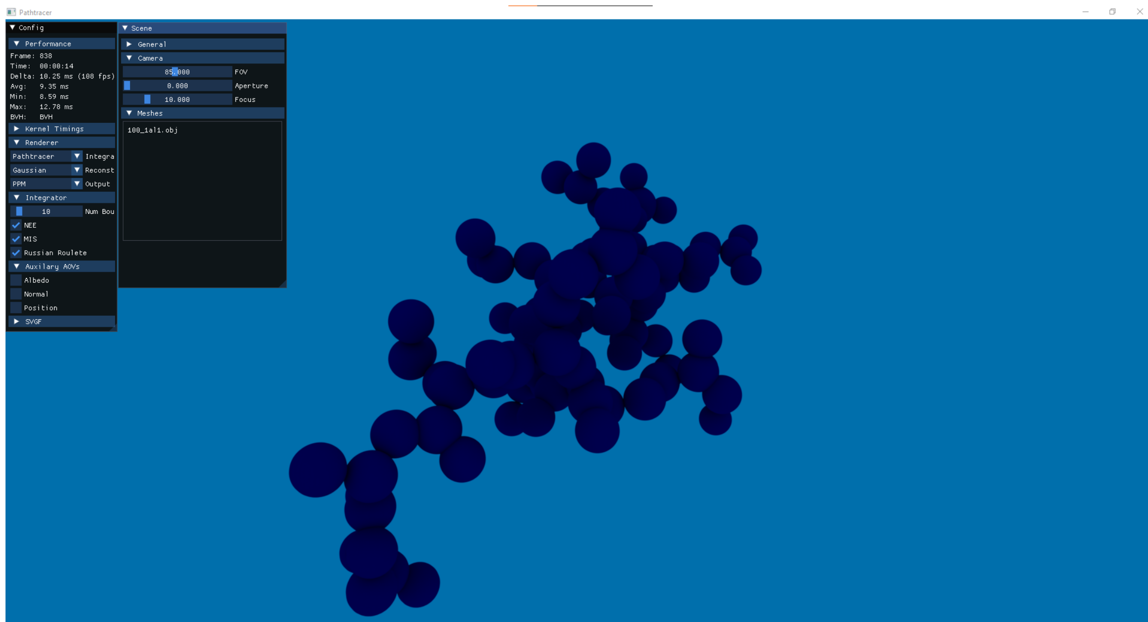

Figure 3.6: Visualization of a protein with 100 atoms, where each atom is represented with a sphere, using the GPU path-tracer.

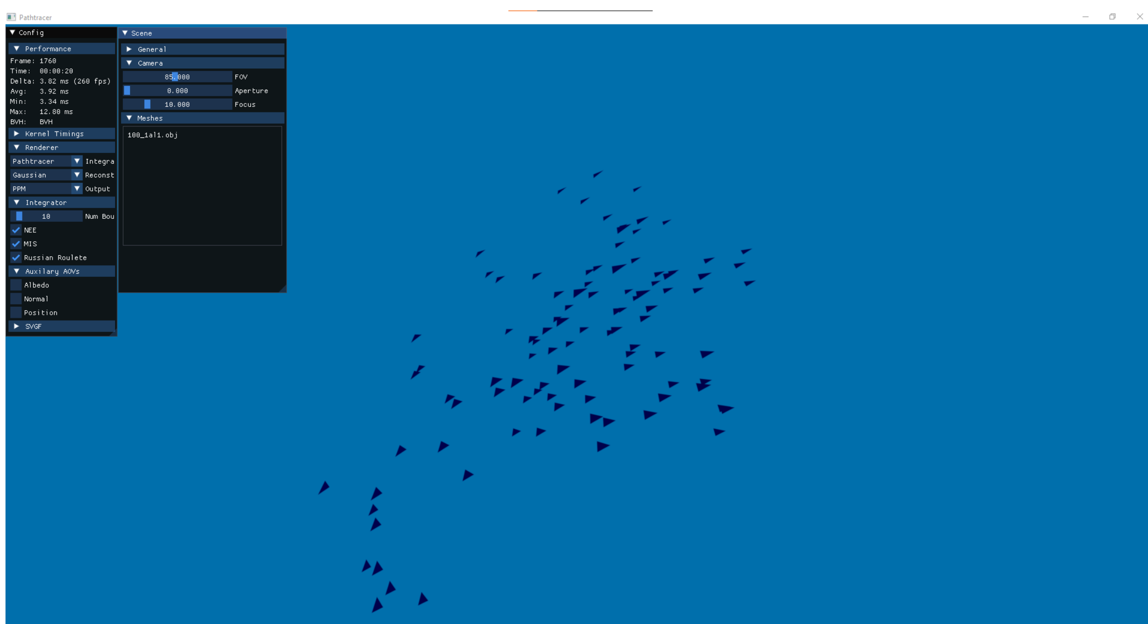

Figure 3.7: Visualization of a protein with 100 atoms, where each atom is represented with a triangle, using the GPU path-tracer.

4

Tests and results

Contents

4.1 Ray tracing tests 39
4.2 Neighbourhood tests 41

In this section, all the used spatial acceleration data structures were tested in order to have a deeper understanding of them. This includes tests that evaluate the efficiency of the data structures, including the time and memory required to execute a certain task. Those tests can be divided into tests regarding ray tracing and tests to find atoms and their neighbours. For all tests, on both CPU and GPU, we used the same set of proteins. All tests were performed on machine with an Intel Core i7-9750H CPU 2.60GHz, 8 GB of RAM, and an NVIDIA GeForce GTX 1660 Ti GPU.

4.1 Ray tracing tests

Ray tracing tests evaluate the efficiency of each of the spatial acceleration data structures to render scenes with different numbers of atoms using a ray tracer algorithm that determines the color of the pixel according to the color of the first object intersected by a ray that was cast from the camera.

Before the comparison between each of the data structures, some other relevant tests were made. One of them uses the parallel OpenMP implementation using BVH with a scene with just a few objects that shows the relation between the number of threads and the average frames per second (FPS) of the scene (fig 4.1). In this case, the FPS increases until around 25 threads, then it remains approximately constant.

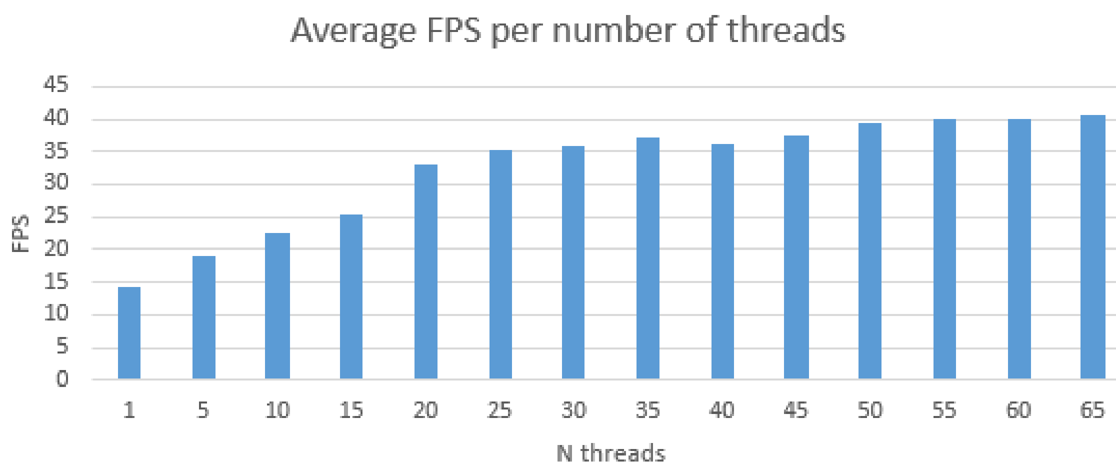

Figure 4.1: Average FPS per number of threads.

Another test (fig 4.2), compares the CPU implementation with the paralleled OpenMP implementation using BVH as data structure. The results show that the paralleled version outperforms the standard CPU implementation in most cases, more specifically in the ones where the scene has more atoms. The parallel approach can achieve a frame rate increase of up to around 40%.

Also, using the OpenMP implementation, it was done a test (fig 4.3), to compare the efficiency of each data structure in the rendering of the scene. The results do not show a significant change between

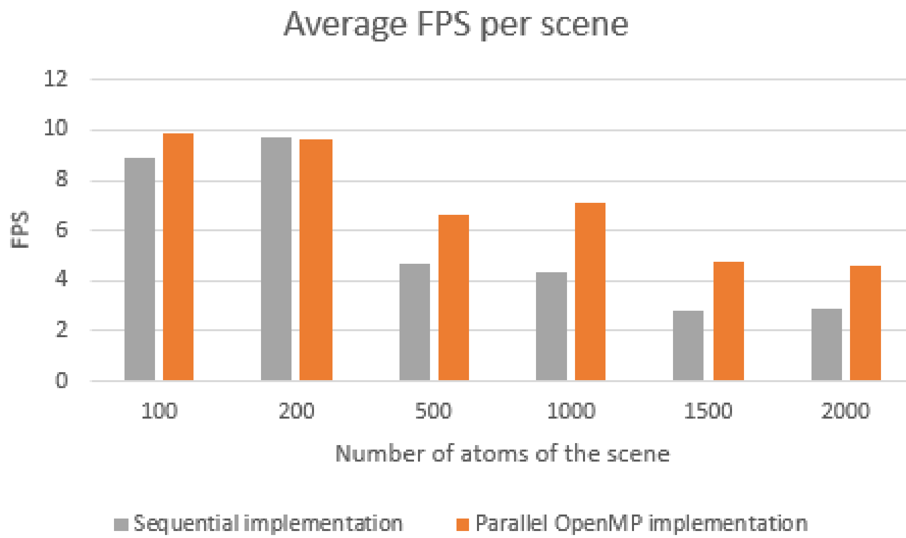

Figure 4.2: Comparison of FPS between a single threaded implementation and the paralleled version.

data structures, although there is usually a slight advantage when using the BVH. Since these tests were not conclusive, it was decided to probe with different tests.

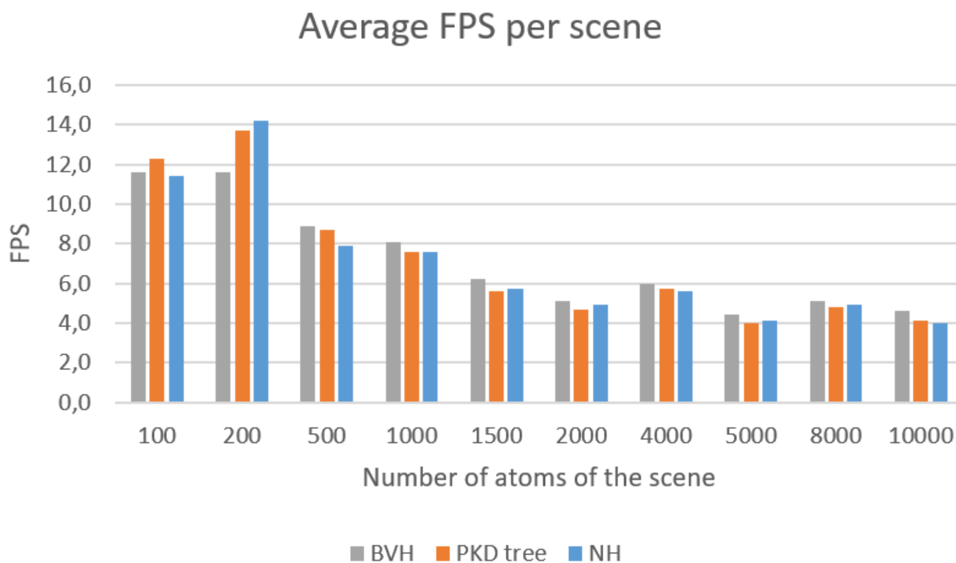

Figure 4.3: Comparison of FPS between data structures in the rendering

4.2 Neighbourhood tests

Neighbourhood tests are used to calculate all neighbours within a certain radius of an atom. That proximity area is determined by the sphere of radius r. In the following tests, the radius used for proximity was 3 Angstroms.

There were many tests involving the calculation of the neighbours where it was gathered the time taken by the task and memory used by the system. Those tests involve either comparing different data structures or changing some variables or thresholds in a specific data structure.

One of the tests involves using the BVH as a data structure and changing the number of atoms that the leaf node would contain, as shown in fig 4.4 and fig 4.5. Results show that, when using a protein with 10000 atoms, the optimal number of atoms per leaf node is around 50 to 100, and in terms of memory, it uses less and less memory as the number of atoms per leaf node increases. This is explained by the fact that this increase leads to a smaller number of total nodes in the data structure, which consequently leads to a decrease in memory use.

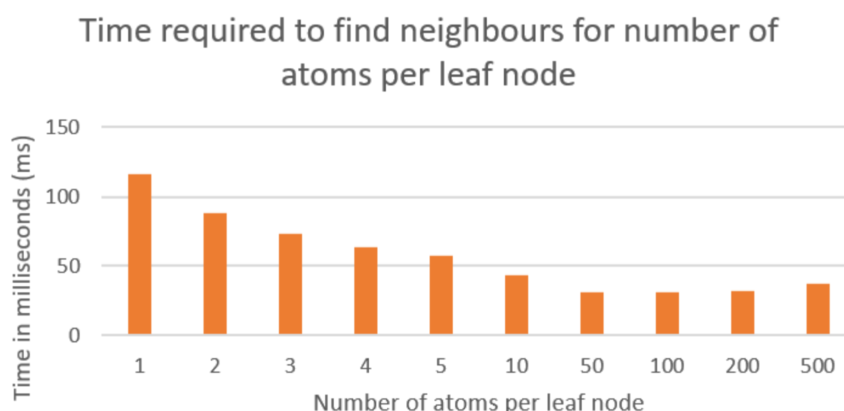

Figure 4.4: Time required to find neighbours for the number of atoms per leaf node.

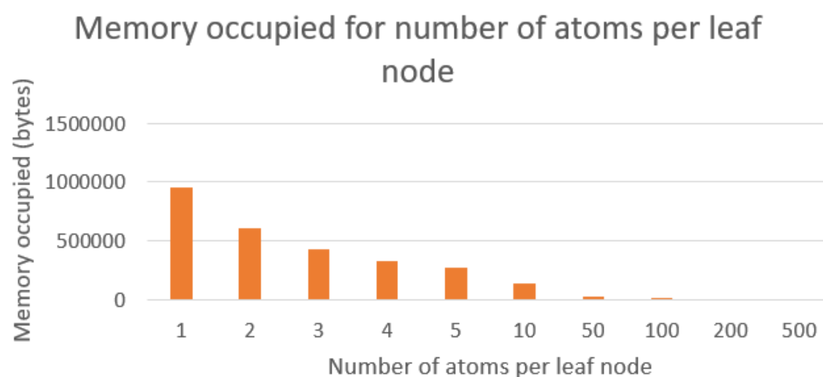

Figure 4.5: Memory occupied for the number of atoms per leaf node.

Other tests involve using the NH as a data structure and changing the value of the threshold, which determines which data structure behavior it will have for each node, as shown in fig 4.6 and fig 4.7. Results show that, using a protein with 10000 atoms, the ideal threshold for both time and memory efficiency is a value in between 500 and 1000. For future tests that do not specify the threshold used in the NH, the value used is a tenth of the number of atoms of the protein.

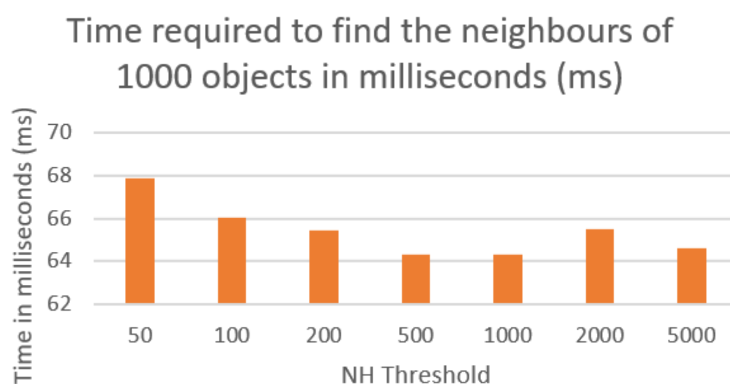

Figure 4.6: Time required to find neighbours for different NH thresholds.

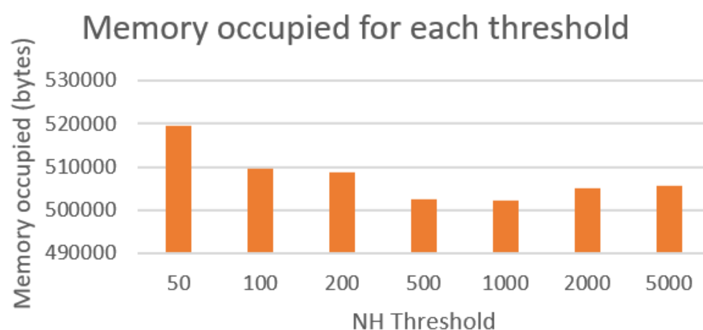

Figure 4.7: Memory occupied for different NH thresholds.

It was also tested, using the BVH GPU implementation and a protein with 10000 atoms, how different sphere radiuses measured in Angstroms would affect the time taken to calculate all the neighbours of the atoms in the scene. As shown in fig 4.8, as the sphere radius is increased, time grows exponentially, as in the first steps the increases are small and later the increases are more significant.

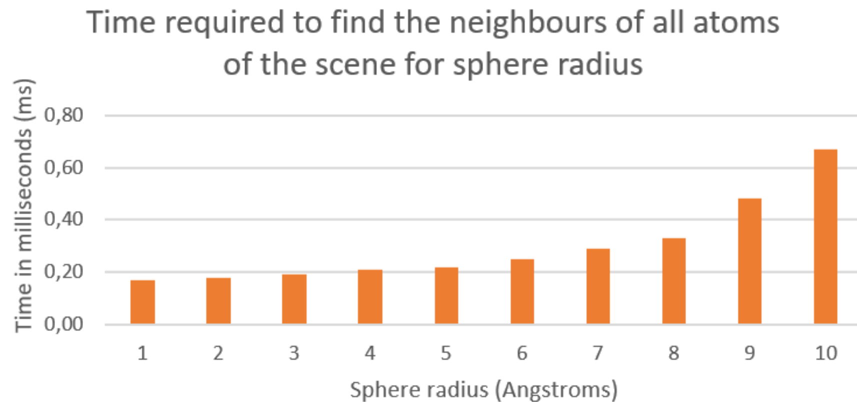

Figure 4.8: Time required to find the neighbours of all atoms of the scene for sphere radius using the BVH GPU.

In the next tests, the comparison between all the spatial acceleration data structures takes place. In fig 4.9 and in more detail in table A.1, it is described the time it takes for each of the data structures to calculate the neighbours of all atoms for each scene. Results show that without using a data structure, column ”NONE”, the time taken grows exponentially as long as the number of atoms in the scene increases. It is also possible to conclude that the best results are, by a long margin, achieved when using the GPU implementations due to the high efficiency of running code. More specifically the BVH GPU is the one with better performance when the number of atoms in the scene is higher than 1000 atoms, and the GPU BVH8 is slightly better than the BVH GPU when the number of atoms is 500 or less. Regarding the implementations in CPU, for both BVH, p-k-d tree and NH, the execution time is in the same order of magnitude, although by analyzing the fig 4.10 and table A.2 it is possible to see that both p-k-d tree and NH can be 6% to 18% faster when compared to BVH. In this case, just one value is used for the nested hierarchy threshold since the results did not differ much.

In fig 4.11 and table A.3, it is shown the memory occupied for each of the data structures for each scene. It is calculated based on the multiplication of the number of nodes in the data structure for the size of each node. Results show that usually the BVH is the one that occupies more memory, followed by a NH with a low value threshold, a NH with a high value threshold, the p-k-d tree, the GPU BVH8 and finally, with the least memory consumption, the BVH GPU implementation.

In fig 4.12 and table A.4, it is shown the percentage increase in efficiency when compared to the CPU BVH, described by a positive percentage, and the decrease in efficiency, described by a negative percentage.

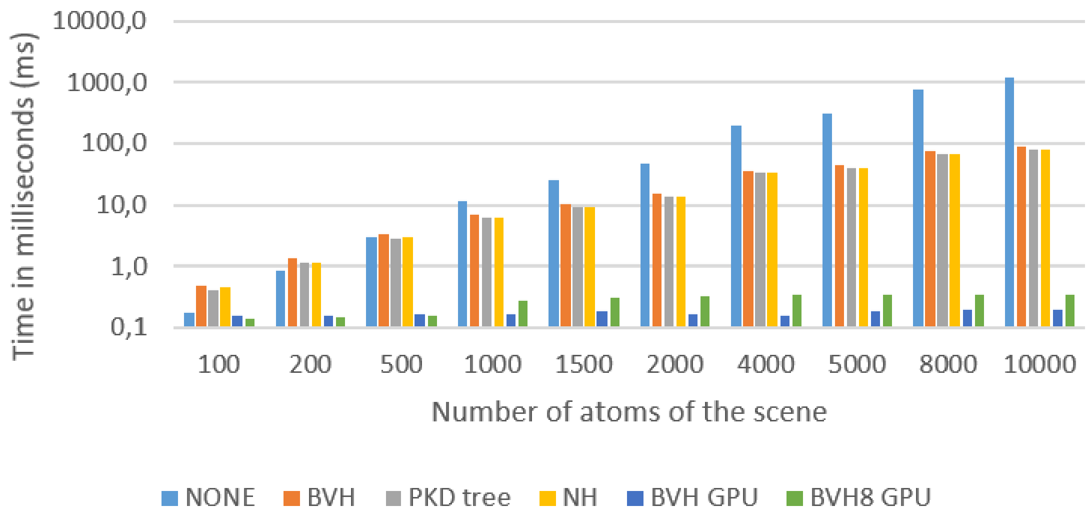

Figure 4.9: Time required to find the neighbours of all atoms of the scene in miliseconds (ms) by each acceleration structure.

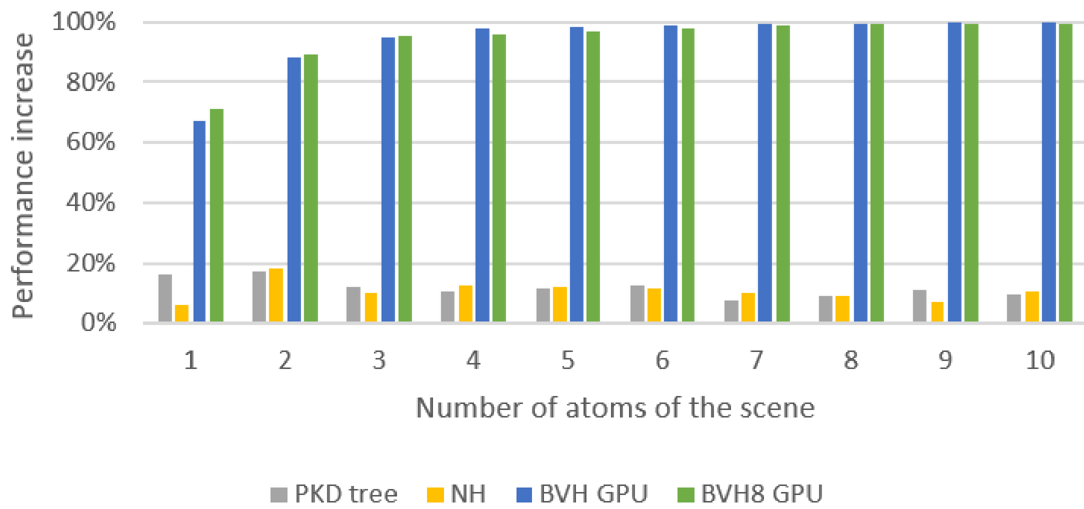

Figure 4.10: Comparison of performance between each acceleration structure and CPU BVH.

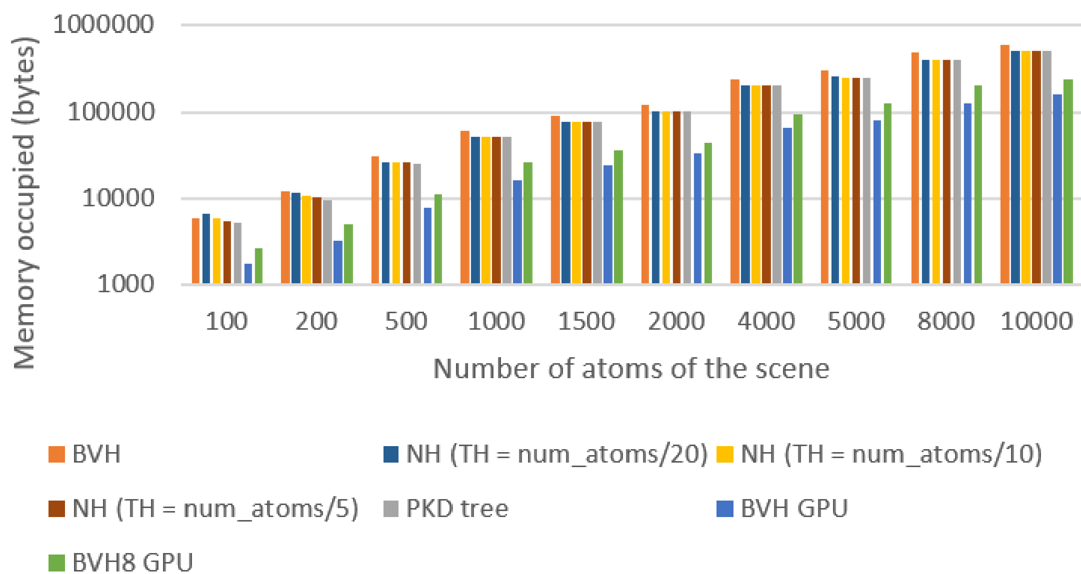

Figure 4.11: Memory required to find the neighbours of all atoms of the scene by each acceleration structure. TH corresponds to the nested hierarchy threshold.

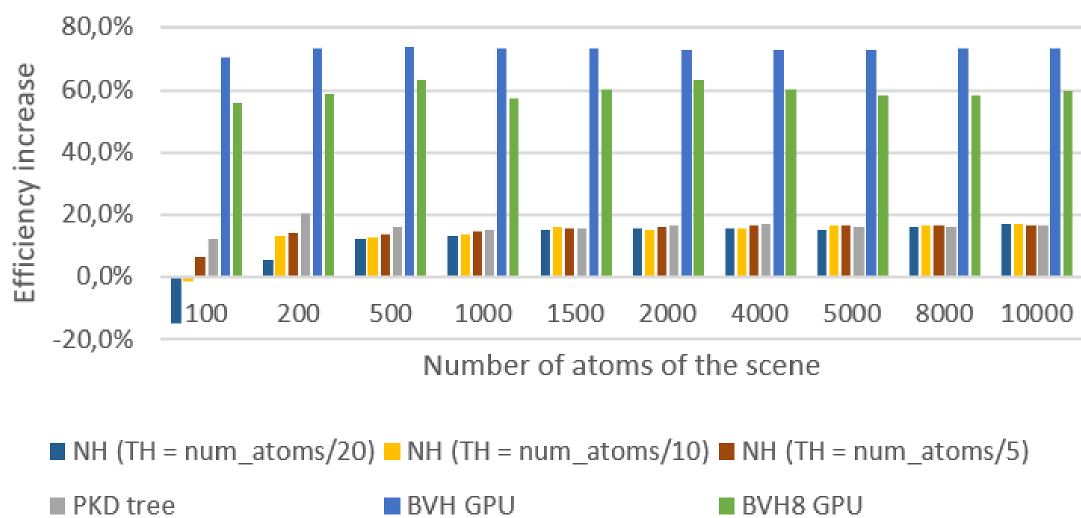

Figure 4.12: Comparison of efficiency between each acceleration structure and CPU BHV. TH corresponds to the nested hierarchy threshold.

5

Conclusion

Contents

5.1 Conclusions 49
5.2 System Limitations and Future Work 49

In this section, a brief summary of the overall work is reported, focusing some of the most relevant findings. It also mentions possible future work as continuation of this study.

5.1 Conclusions

The aim of this research was to compare the performance of different acceleration structures. First, it was applied to ray tracing of a scene and then to the neighbourhood calculation of atoms of proteins, mainly by using the distance between particles and the algorithm to traverse the data structure described earlier. This neighbourhood calculation allowed a reduction in the number of pairwise interactions to be calculated in a molecular system, leading to a faster calculation of the total energy of the force field generated by the atoms. In this document, it was explained how this energy calculation is done, as well as the main motivations and limitations for the implementation of acceleration structures. It was explained how some of the other implementations work to speed up this heavy computation, including the NH solution, a hybrid between the BVH and the p-k-d tree.

Regarding ray tracing in CPU, results show that the parallel OpenMP implementation can achieve a 40% increase in frame rate when compared to the sequential implementation. When using the NH implementation, in our case, the optimal NH threshold is around a tenth of the number of atoms of the protein. Regarding the neighbourhood calculation in the CPU, both the p-k-d tree and the NH outperform the BVH and require less memory. The GPU implementations are more efficient and require less memory than all the CPU implementations, where the GPU BVH is the one that, in general, performs better for the neighbourhood calculation.

5.2 System Limitations and Future Work

In the GPU implementations, we used a top-down approach in the construction of the BVH. Although the LBVH uses a bottom-up approach using Morton codes that, according to Karras [21], allows more parallelism, due to time constraints, it was chosen to use a top-down approach since it was the same construction algorithm as our CPU implementation, so it was possible to use the same algorithm to traverse the data structure. This allows a better comparison between the CPU and GPU implementations.

In the future, it would be interesting to continue this work by implementing this proposed nested hierarchy on the GPU, either by using CUDA, OpenMP, or OpenACC, and do a comparative analysis with the GPU BVH of both efficiency and memory consumption. Additionally, this nested hierarchy implementation in GPU could be integrated in a molecular dynamics simulation tool like the ones aforementioned, which calculate the interactions between atoms such as the calculation of the energy of the system by applying the equations referred in the section 2.2. This would also allow a comparison between this

new methodology and the approaches that these tools are using. It would be particularly relevant to do the comparison with the LBVH implementation [33] present in the simulation tool HOOMD-blue, which is based on Karras's construction algorithm [21].

Bibliography

[1] M. P. Allen, “Introduction to molecular dynamics simulation introduction,” *Computational Soft Matter: From Synthetic Polymers to Proteins, Lecture Notes*, vol. 23, 2004.

[2] C.-e. A. Chang, Y.-m. M. Huang, L. J. Mueller, and W. You, “Investigation of structural dynamics of enzymes and protonation states of substrates using computational tools,” *Catalysts*, vol. 6, no. 6, 2016. [Online]. Available: https://www.mdpi.com/2073-4344/6/6/82

[3] D. J. Hardy, Z. Wu, J. C. Phillips, J. E. Stone, R. D. Skeel, and K. Schulten, “Multilevel summation method for electrostatic force evaluation,” *Journal of Chemical Theory and Computation*, vol. 11, pp. 766–779, 2 2015.

[4] B. Li, “A comparative analysis of spatial partitioning methods for large-scale , real-time crowd simulation,” 2014.

[5] S. Raschdorf and M. Kolonko, “A comparison of data structures for the simulation of polydisperse particle packings,” *International Journal for Numerical Methods in Engineering*, vol. 85, pp. 625–639, 2 2011.

[6] M. P. Howard, A. Statt, F. Madutsa, T. M. Truskett, and A. Z. Panagiotopoulos, “Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units,” *Computational Materials Science*, vol. 164, pp. 139–146, 6 2019.

[7] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, “Fast bvh construction on gpus,” 2009.

[8] H. M. Berman, “The protein data bank,” *Nucleic Acids Research*, vol. 28, pp. 235–242, 1 2000.

[9] K. Binder, J. Horbach, W. Kob, P. Wolfgang, and V. Fathollah, “Molecular dynamics simulations,” *Journal of Physics Condensed Matter*, vol. 16, 2 2004.

[10] L. Monticelli and D. P. Tieleman, “Force fields for classical molecular dynamics,” pp. 197–213, 2013.

[11] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz, D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman, “A second generation force field for the simulation of proteins, nucleic acids, and organic molecules,” *Journal of the American Chemical Society*, vol. 118, pp. 2309–2309, 1 1996.

[12] A. D. MacKerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiórkiewicz-Kuczera, D. Yin, and M. Karplus, “All-atom empirical potential for molecular modeling and dynamics studies of proteins,” *The Journal of Physical Chemistry B*, vol. 102, pp. 3586–3616, 4 1998.

[13] C. Oostenbrink, A. Villa, A. E. Mark, and W. F. V. Gunsteren, “A biomolecular force field based on the free enthalpy of hydration and solvation: The gromos force-field parameter sets 53a5 and 53a6,” *Journal of Computational Chemistry*, vol. 25, pp. 1656–1676, 10 2004.

[14] W. L. Jorgensen, D. S. Maxwell, and J. Tirado-Rives, “Development and testing of the opls all-atom force field on conformational energetics and properties of organic liquids,” *Journal of the American Chemical Society*, vol. 118, pp. 11 225–11 236, 11 1996.

[15] T. Darden, D. York, and L. Pedersen, “Particle mesh ewald: An n log (n) method for ewald sums in large systems,” *The Journal of Chemical Physics*, vol. 98, pp. 10 089–10 092, 6 1993.

[16] R. Salomon-Ferrer, A. W. Götz, D. Poole, S. L. Grand, and R. C. Walker, “Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald,” *Journal of Chemical Theory and Computation*, vol. 9, pp. 3878–3888, 9 2013.

[17] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, “A smooth particle mesh ewald method,” *The Journal of Chemical Physics*, vol. 103, pp. 8577–8593, 1995.

[18] P. J. in ’t Veld, S. J. Plimpton, and G. S. Grest, “Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics,” *Computer Physics Communications*, vol. 179, pp. 320–329, 9 2008.

[19] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka, “Cpu ray tracing large particle data with balanced p-k-d trees.” IEEE, 10 2015, pp. 57–64.

[20] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, “Efficient collision detection using bounding volume hierarchies of k-dops,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, pp. 21–36, 1998.

[21] T. Karras, “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees,” 2012, pp. 33–37.

[22] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies.” ACM Press, 2013, p. 89.

[23] M. Vinkler, V. Havran, and J. Bittner, “Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing,” *Computer Graphics Forum*, vol. 35, pp. 68–79, 12 2016.

[24] M. Vinkler, J. Bittner, and V. Havran, “Extended morton codes for high performance bounding volume hierarchy construction.” Association for Computing Machinery, Inc, 7 2017.

[25] J. Pantaleoni and D. Luebke, “Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry,” 2010. [Online]. Available: http://code.google.

[26] A. Pérard-Gayot, J. Kalojanov, and P. Slusallek, “Gpu ray tracing using irregular grids,” *Computer Graphics Forum*, vol. 36, pp. 477–486, 5 2017.

[27] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, “Fast, effective bvh updates for animated scenes.” ACM Press, 2012, p. 197.

[28] J. Hendrich, D. Meister, and J. Bittner, “Parallel bvh construction using progressive hierarchical refinement,” 2017.

[29] P. Gralka, I. Wald, S. Geringer, G. Reina, and T. Ertl, “Spatial partitioning strategies for memory-efficient ray tracing of particles.” Institute of Electrical and Electronics Engineers Inc., 10 2020, pp. 42–52.

[30] J. A. Anderson, J. Glaser, and S. C. Glotzer, “Hoomd-blue: A python package for high-performance molecular dynamics and hard particle monte carlo simulations,” *Computational Materials Science*, vol. 173, 2 2020.

[31] J. A. Anderson, M. E. Irrgang, and S. C. Glotzer, “Scalable metropolis monte carlo for simulation of hard shapes,” *Computer Physics Communications*, vol. 204, pp. 21–30, 7 2016.

[32] D. Frenkel and B. Smit, *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2001, vol. 1.

[33] M. P. Howard, J. A. Anderson, A. Nikoubashman, S. C. Glotzer, and A. Z. Panagiotopoulos, “Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units,” *Computer Physics Communications*, vol. 203, pp. 45–52, 6 2016.

[34] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - short range forces," *Computer Physics Communications*, vol. 182, pp. 898–911, 4 2011.

[35] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - particle-particle particle-mesh," *Computer Physics Communications*, vol. 183, pp. 449–459, 3 2012.

[36] T. W. Sirk, S. Moore, and E. F. Brown, "Characteristics of thermal conductivity in classical water models," *Journal of Chemical Physics*, vol. 138, 2 2013.

[37] J. C. Phillips, D. J. Hardy, J. D. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang, R. McGreevy, M. C. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. Kalé, K. Schulten, C. Chipot, and E. Tajkhorshid, "Scalable molecular dynamics on cpu and gpu architectures with namd," *Journal of Chemical Physics*, vol. 153, 7 2020.

[38] J. E. Stone, A.-P. Hynninen, J. C. Phillips, and K. Schulten, "Early experiences porting the namd and vmd molecular simulation and analysis software to gpu-accelerated openpower platforms," pp. 188–206, 2016.

[39] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl, "Tackling exascale software challenges in molecular dynamics simulations with gromacs," pp. 3–27, 2015.

[40] C. L. Wennberg, T. Murtola, B. Hess, and E. Lindahl, "Lennard-jones lattice summation in bilayer simulations has critical effects on surface tension and lipid properties," *Journal of Chemical Theory and Computation*, vol. 9, pp. 3527–3537, 8 2013.

[41] C. L. Wennberg, T. Murtola, S. Páll, M. J. Abraham, B. Hess, and E. Lindahl, "Direct-space corrections enable fast and accurate lorentz-berthelot combination rule lennard-jones lattice summation," *Journal of Chemical Theory and Computation*, vol. 11, pp. 5737–5746, 11 2015.

[42] P. Eastman and V. S. Pande, "Constant constraint matrix approximation: A robust, parallelizable constraint method for molecular simulations," *Journal of Chemical Theory and Computation*, vol. 6, pp. 434–437, 2 2010.

[43] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery, "Arborx: A performance portable geometric search library," *ACM Transactions on Mathematical Software*, vol. 47, 1 2021.

[44] M. Robinson and M. Bruna, "Particle-based and meshless methods with aboria," *SoftwareX*, vol. 6, pp. 172–178, 2017.

[45] S. Guntury and P. J. Narayanan, “Raytracing dynamic scenes on the gpu using grids,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 5–16, 2012.

[46] V. Ogarko and S. Luding, “A fast multilevel algorithm for contact detection of arbitrarily polydisperse objects,” *Computer Physics Communications*, vol. 183, pp. 931–936, 4 2012.

[47] D. Krijgsman, V. Ogarko, and S. Luding, “Optimal parameters for a hierarchical grid data structure for contact detection in arbitrarily polydisperse particle systems,” *Computational Particle Mechanics*, vol. 1, pp. 357–372, 9 2014.

[48] D. Bautembach, “Animated sparse voxel octrees,” *Bachelors Thesis, Univeristy Of Hamburg*, 2011.

[49] B. Mados, E. Chovancova, and M. Hasin, “Evaluation of pointerless sparse voxel octrees encoding schemes using huffman encoding for dense volume datasets storage.” Institute of Electrical and Electronics Engineers Inc., 11 2020, pp. 424–430.

[50] B. Mados, N. Adam, and M. Stancel, “Representation of dense volume datasets using pointerless sparse voxel octrees with variable and fixed-length encoding.” Institute of Electrical and Electronics Engineers Inc., 1 2021, pp. 343–348.

[51] H. Ylitie, T. Karras, and S. Laine, “Efficient incoherent ray traversal on gpus through compressed wide bvhs,” in *Proceedings of High Performance Graphics*, 2017, pp. 1–13.

[52] J. Bergen, “Gpu-raytracer,” https://github.com/jan-van-bergen/GPU-Raytracer, 2022, accessed 31 July, 2022.

Results Tables

Table A.1: Time required to find the neighbours of 1000 objects in milliseconds (ms) by each acceleration structure.

Number of atoms	NONE	BVH	PKD tree	NH	GPU BVH	GPU BVH8
100	0,2	0,5	0,4	0,5	0,16	0,14
200	0,9	1,4	1,1	1,1	0,16	0,15
500	3,0	3,3	2,9	3,0	0,17	0,16
1000	11,5	7,1	6,4	6,2	0,17	0,28
1500	26,1	10,6	9,3	9,3	0,19	0,31
2000	48,5	16,0	13,9	14,1	0,17	0,33
4000	199,4	37,2	34,4	33,4	0,16	0,36
5000	305,6	44,6	40,5	40,5	0,19	0,36
8000	787,0	74,7	66,1	69,3	0,20	0,36
10000	1230,7	90,1	81,5	80,6	0,20	0,36

Table A.2: Comparison of performance between each acceleration structure and CPU BVH.

Number of atoms	None	PKD tree	NH	GPU BVH	GPU BVH8
100	63,3%	16,3%	6,1%	67,3%	71,4%
200	35,5%	17,4%	18,1%	88,4%	89,1%
500	9,7%	12,1%	10,3%	94,9%	95,2%
1000	-61,0%	10,5%	12,8%	97,6%	96,1%
1500	-146,9%	11,8%	12,2%	98,2%	97,1%
2000	-203,6%	12,8%	12,0%	98,9%	97,9%
4000	-436,4%	7,6%	10,2%	99,6%	99,0%
5000	-585,9%	9,1%	9,0%	99,6%	99,2%
8000	-953,9%	11,5%	7,2%	99,7%	99,5%
10000	-1265,6%	9,5%	10,6%	99,8%	99,6%

Table A.3: Memory required to find the neighbours of all atoms of the scene by each acceleration structure. N is the number of atoms of the scene and TH corresponds to the nested hierarchy threshold.

N	BVH	NH (TH = N/20)	NH (TH = N/10)	NH (TH = N/5)	PKD tree	GPU BVH	GPU BVH8
100	5808	6664	5880	5432	5096	1728	2560
200	12048	11368	10472	10360	9576	3200	4960
500	30000	26376	26152	25928	25144	7872	11040
1000	59952	51912	51576	51240	50904	15936	25680
1500	90192	76552	75768	76216	76104	24064	35600
2000	120528	101416	102424	100856	100408	32384	44480
4000	241200	203672	203336	201320	200200	64832	95920
5000	300144	253960	250824	250264	251608	80960	124720
8000	482160	404376	402696	402472	404824	127328	201520
10000	606288	502600	502264	505064	506184	159936	242320

Table A.4: Comparison of efficiency between each acceleration structure and CPU BHV. N is the number of atoms of the scene and TH corresponds to the nested hierarchy threshold.

N	NH (TH = N/20)	NH (TH = N/10)	NH (TH = N/5)	PKD tree	GPU BVH	GPU BVH8
100	-14,7%	-1,2%	6,5%	12,3%	70,2%	55,92%
200	5,6%	13,1%	14,0%	20,5%	73,4%	58,83%
500	12,1%	12,8%	13,6%	16,2%	73,8%	63,20%
1000	13,4%	14,0%	14,5%	15,1%	73,4%	57,17%
1500	15,1%	16,0%	15,5%	15,6%	73,3%	60,53%
2000	15,9%	15,0%	16,3%	16,7%	73,1%	63,10%
4000	15,6%	15,7%	16,5%	17,0%	73,1%	60,23%
5000	15,4%	16,4%	16,6%	16,2%	73,0%	58,45%
8000	16,1%	16,5%	16,5%	16,0%	73,6%	58,20%
10000	17,1%	17,2%	16,7%	16,5%	73,6%	60,03%