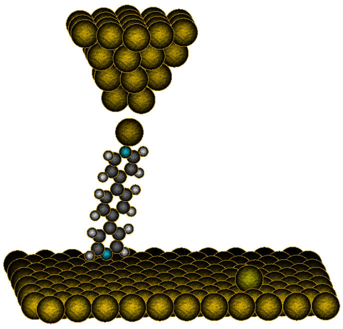
Real time force field simulation for stm controlled molecular electronics experiments using Cuda gpu programming



Thesis

submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE
in
PHYSICS AND COMPUTER SCIENCE

Author :	Dyon van Vreumingen
Student id :	s1348434
Supervisors :	Prof. dr. J.M. van Ruitenbeek Dr. ir. F.J. Verbeek Msc. S. Tewari

Leiden, The Netherlands, August 28, 2016

Real time force field simulation for stm controlled molecular electronics experiments using Cuda gpu programming

Dyon van Vreumingen

Huygens-Kamerlingh Onnes laboratory, Leiden university
P.O. Box 9500, 2300 RA Leiden, The Netherlands

August 28, 2016

ABSTRACT

Since the introduction of the *scanning tunnelling microscope* (stm) as a tool for experiments in nanotechnology, much research has been done on topics such as molecular electronics and atomic surfaces. Recently, a program was developed that simulates the motion of a system consisting of a gold substrate and gold adatoms, under influence of an stm tip, through Newtonian molecular dynamics. Since the program was aimed at assisting stm experiments, it needed the ability to simulate a sample with a tip in real-time, thus demanding high performance. In this project, we work on the extension of the simulation to a new type of stm experiment, which involves an organic molecule in addition to the gold elements. The more complex nature of such systems, however, makes these simulation severely more computation heavy; this is a serious problem that has to be addressed. In our quest of speeding up the program, we reimplement certain algorithms on a graphics processing unit (gpu), with the Cuda gpu programming framework.

Keywords: scanning tunnelling microscope, molecular dynamics simulation, general purpose gpu

Contents

Chapter 1

Introduction

Over the past years, much progress has been made in the field of atomic and molecular electronics. Since its first proposal in 1974[1], many articles have been published, describing research exploring its nature and applicability. However, there is a clear discrepancy in the suggested models and reported experiments. Jan van Ruitenbeek's research group, where I have been working on my bachelor research project, seeks to bring these closer together by developing tools for conducting these types of experiments in a more standardised fashion.

During the spring of 2014, bachelor student Jacob Bakermans, in collaboration with S. Tewari and C. Wagner, aided this research by developing a computer program that provides a *real-time* simulation of scanning tunnelling microscopy (stm) experiments, taking the movement of the actual stm tip as input[2]. His program is based on the molecular dynamics paradigm of atomic and molecular simulation[3], and mimics the behaviour of gold atoms by modelling their interactions with Newtonian mechanics.

The task we face now is to extend this simulation program to other types of stm experiments; the original program, after all, was written for experiments featuring gold adatoms placed on gold substrates. The experiments that are currently either being carried out or planned for a future stadium of the research require the simulation program to support certain molecules (bipyridyl benzene, in this case) on a gold substrate, in addition to gold adatoms.

This, however, drastically complicates matters. For one, many new interactions—that is, interactions between gold and molecule atoms, as well as interactions between molecule atoms themselves—are introduced, and these all have to be modelled properly. Secondly, atomic vibrations in the molecule and the gold atoms need to be reconsidered. It is mainly the molecule atoms that tend to vibrate at a higher frequency than the gold atoms which pose certain problems, slowing down the simulation, to the point that it can no more run in real-time.

In this bachelor research project, we seek to complete this extension of the program to a gold-molecule system, which has partially been developed already, primarily by looking for ways to make the program faster. This acceleration is done with the aid of graphics card programming, using the Nvidia Cuda platform. All aspects of the design and implementation of the gpu algorithms, along with a theoretical reference framework, and an elaboration of the issues we seek to address are discussed in this thesis.

Chapter 2

Theoretical background

2.1 Principles of molecular dynamics

Molecular dynamics is a well-known and widely employed method for simulating ensembles at microscopic level in a Newtonian fashion[3]. During the process of simulation, the atoms in the ensemble are constantly driven to move by interaction forces, as dictated by classical equations of motion. These interaction forces have been derived beforehand from continuous potentials, which were obtained through empirical and quantum mechanics studies of these systems. This structure makes the simulation entirely dependent on the choice of correct potentials for modelling the atoms, as these contain the only input for behavioural information. As such, they form the entry point for any quantum mechanical modelling in an otherwise classical representation. This is different from density functional theory (dft), where the quantum mechanical nature of the atomic arrangement is directly taken into accountthe by using the electron density, which serves as a basis for describing the ground state of a many-body system[4, 5]. However, we are only interested in recording atomic motion (charge transport plays merely an implicit role here), and since the simulation was developed to run in real-time together with stm experiments, we cannot use dft calculations that tend to take hours or even days to complete.

Once the forces exerted on each atom are known, the velocities and the displacements of these atoms are determined through numerical integration. For this purpose, the *velocity-Verlet* integration algorithm[6] is used. This algorithm is based on the second-order series expansion of the position $\mathbf{r}_i$ of particle i about time t, with a deviation Δt:

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \dot{\mathbf{r}}_i(t)\Delta t + \tfrac{1}{2}\ddot{\mathbf{r}}_i(t)\Delta t^2; \tag{2.1}$$

the corresponding velocity $\dot{\mathbf{r}}_i(t+\Delta t)$ may then be expressed as

$$\dot{\mathbf{r}}_i(t+\Delta t) = \dot{\mathbf{r}}_i(t) + \tfrac{1}{2}(\ddot{\mathbf{r}}_i(t) + \ddot{\mathbf{r}}_i(t+\Delta t))\Delta t. \tag{2.2}$$

It is immediately clear that the acceleration $\ddot{\mathbf{r}}_i(t)$ can be derived from the force $\mathbf{F}_i(t)$ via Newton's second law. Coming back to the principle of iteratively moving the atoms, we can regard the deviation Δt as the time that passes between two iterations—the so-called *timestep*. Choosing the right timestep is critical for producing a stable system that shows correct atomic behaviour. In theory, we would like our timestep to be infinitesimal, thus reducing the error to naught; however, the smaller the timestep, the less simulation time elapses per second. Since we wish to see the evolution of the system in real-time, and the calculation speed is limited by the hardware, we must accept a tiny error caused by a finite timestep. The matter of timesteps is further discussed in chapter 3.

The last step of a each iteration is the application of a *thermostat*. The task of this thermostat is to artificially dissipate excess energy that is brought externally into the system. In our case, this energy comes from the tip, which is given an additional velocity—and thus an additional kinetic energy, which it can pass over to the other atoms—with the input device. The reason for the necessity of a thermostat is twofold:

1 if there is no way for the energy to leave the system, atomic vibrations will never be damped, which will eventually destabilise the substrate and the tip;

2 in the stm experiments that are conducted in reality, the temperature is kept constant, and we want to ensure this is also the case in the simulation[2].

For keeping the virtual temperature at a constant value, we cannot choose any arbitrary thermostat. Several algorithms have been proposed for this purpose[7, 8], which usually revolve around the description of temperature in statistical physics, taking into account kinetic energy of the atoms, and the equipartition theorem. The mechanism for drawing away energy is a *friction factor* which rescales all velocities in each step, thus reducing the total kinetic energy. Regrettably, we did not have enough time to reimplement one of these thermostats on the gpu, and for this reason, we decided to use a *constant* friction factor.

2.2 Interactions between gold atoms

With respect to the old simulation, the potential of gold-gold interaction and the derived force remain unchanged. We use the following potential[9, 10]:

$$V = -\zeta \sum_{i}^{N} \sqrt{\sum_{j \neq i}^{N} e^{-2q(r_{ij}/r_0 - 1)}} + A \sum_{i}^{N} \sum_{j \neq i}^{N} e^{-p(r_{ij}/r_0 - 1)}, \qquad (2.3)$$

where ζ, A, p, q and r_0 are empirically determined constants, and r_{ij} is the scalar distance between atom i and j. This potential consists of an attractive part (the first term), and a repulsive part (the second term). While the combination of these two terms produces a curve similar in shape to that of a generic Lennard-Jones potential, it originates from dynamics of the conduction electrons and the band structure of the lattice[10, 11].

The force that results from this takes the following form[2]:

$$\begin{aligned} \mathbf{F}_i = & -\frac{q\zeta}{r_0} \frac{1}{s_i} \sum_{j \neq i}^{N} \frac{(\mathbf{r}_i - \mathbf{r}_j)}{r_{ij}} e^{-2q(r_{ij}/r_0 - 1)} \\ & - \frac{q\zeta}{r_0} \sum_{j \neq i}^{N} \frac{1}{s_j} \frac{(\mathbf{r}_i - \mathbf{r}_j)}{r_{ij}} e^{-2q(r_{ij}/r_0 - 1)} \\ & + \frac{2Ap}{r_0} \sum_{j \neq i}^{N} \frac{(\mathbf{r}_i - \mathbf{r}_j)}{r_{ij}} e^{-p(r_{ij}/r_0 - 1)}, \end{aligned} \qquad (2.4)$$

where

$$s_i = \sqrt{\sum_{j \neq i}^{N} e^{-2q(r_{ij}/r_0 - 1)}}. \qquad (2.5)$$

2.3 Gold-molecule interactions

At the current moment, we use generic Van der Waals forces to model the interplay between gold atoms and most of the molecule atoms. We chose to implement the following Lennard-Jones potential for this purpose:

$$V_{ij}^{\mathrm{LJ}} = \epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]. \qquad (2.6)$$

This is a straightforward and widely used method to obtain reasonably accurate Van der Waals forces, and is also relatively fast to calculate. After all, only one division and three multiplications need to be carried out in order to compute the attractive part (which is the fraction raised to the power 6) and one more multiplication to find the repulsive part.

It appears that the lowermost atom of the stm tip, also dubbed the apex atom, is much more reactive in the vicinity of nitrogen atoms. Therefore, we felt it was necessary to add another potential in order to take this into account. We decided to let this potential be a stronger Morse function:

$$V_{ij}^{\text{Morse}} = D(1 - e^{-\alpha(r_{ij} - r_0)})^2. \tag{2.7}$$

2.4 Intramolecular interactions

For the last part of the interactions, the forces between atoms in the molecule, we adopt the model used by Cornell et al[12]. This model consists of a number of harmonic oscillator potentials that keep the molecule together as a whole. Since the atoms are not charged and no electric field is present, we omitted the Coulumb potential that was employed in their research.

The first and simplest of the three is the bond potential, which describes the spring-mass system like behaviour of two bonded atoms. This function is defined as follows:

$$V_{ij}^{\text{bond}} = \tfrac{1}{2} k_{ij}^{\text{bond}} (r_{ij} - r_0)^2. \tag{2.8}$$

Here, r_{ij} is the distance between two atoms i and j, as shown in figure 2.1[13], and k_{ij}^{bond} is a bond constant that depends on the types of the atoms that are involved in the covalent bond. The corresponding force on atom i is equal in value to the force on atom j—albeit in the other direction—and is easily derived:

$$\mathbf{F}_i^{\text{bond}} = -\frac{\partial}{\partial \mathbf{r}_i} V_{ij}^{\text{bond}} = -k_{ij}^{\text{bond}} (r_{ij} - r_0) \frac{\mathbf{r}_{ij}}{r_{ij}}. \tag{2.9}$$

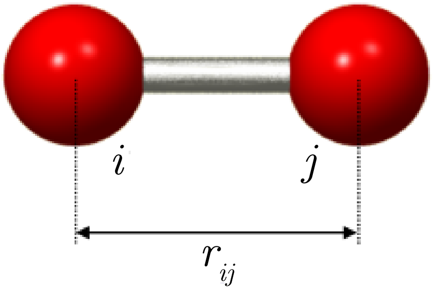


Figure 2.1. *Two bonded atoms i and j, with interatomic distance r_{ij}.*

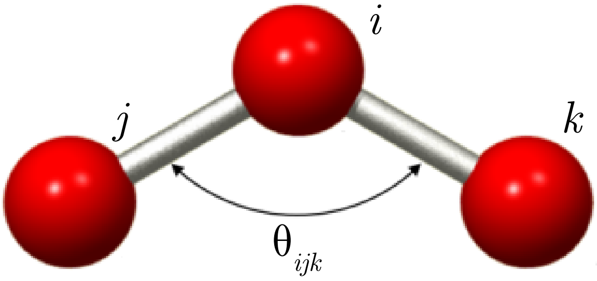


Figure 2.2. *An angle group with atoms i, j, k and the angle θ_{ijk} between these three atoms. From the definition of the angle in equation 2.11, it must be that $0 \leq \theta_{ijk} \leq \pi$.*

Secondly, we have the angle potential. When three bonded atoms form an angular group, as shown in figure 2.2[13], the lowest energy configuration appears to occur when the angle between the three atoms, where the middle atom acts as a pivot, is equal to some equilibrium angle θ_0. This is expressed in the following equation:

$$V_{ijk}^{\text{angle}} = k_{ijk}^{\text{angle}}(\cos\theta_{ijk} - \cos\theta_0)^2, \tag{2.10}$$

where

$$\cos\theta_{ijk} = \frac{\mathbf{r}_{ij} \bullet \mathbf{r}_{ik}}{r_{ij}r_{ik}}. \tag{2.11}$$

Please note that $\mathbf{r}_{ab}$ denotes the vector from a to b, and is thus equal to $\mathbf{r}_b - \mathbf{r}_a$. The force on the middle atom i, then, is different from that exerted on the outer atoms j and k; this is further discussed in section 4.7.1.

Lastly, we need to consider the dihedral potential, which emerges from the strain put on a molecule when a group of atoms which would lie in one plane in the ground state, are bent out of this plane. Such groups are named dihedral groups, and consist of four atoms. This is illustrated in figure 2.3[13]. The planes that the four atoms lie in, can be described by their respective normal vectors:

$$\mathbf{u} = \mathbf{r}_{ij} \times \mathbf{r}_{ik}, \qquad \mathbf{v} = \mathbf{r}_{kl} \times \mathbf{r}_{ik}, \tag{2.12}$$

and the angle ϕ_{ijkl} between these two planes is then defined by

$$\cos\phi_{ijkl} = \frac{\mathbf{u} \bullet \mathbf{v}}{uv}. \tag{2.13}$$

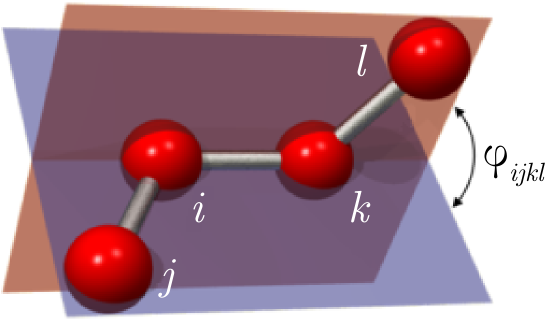


Figure 2.3. *A dihedral group whose atoms i, j and k lie in the blue plane, while atoms i, k and l lie in the orange plane. The force that results from the dihdral potential tends to bring these two planes together to form one single plane.*

The final potential depends on this angle in the following way:

$$V_{ijkl}^{\text{dihedral}} = k_{ijkl}^{\text{dihedral}}(1 + \cos(2\phi_{ijkl} - \phi_0)). \tag{2.14}$$

Similar to the case of the angular potential, the outermost atoms, which are bonded to only one atom in the group, need other treatment in terms of forces than those that are bonded to two atoms in the group. How we deal with this discrepancy is set out in section 4.7.2.

Chapter 3

Project goal

3.1 Simulation of stm manipulation

Scanning tunnelling microscopy is a technique that was developed for imaging structures at atomic scales[14]. Images are obtained by bringing an atomically sharp tip very close (of the order of nanometres or even Ångströms) to a surface, and measuring the *tunnelling current* that starts flowing between the sample and the tip. Later it was discovered that this device is also capable of moving adatoms, which are atoms lying on top of the sample surface, either by pulling or pushing them around[15]. Doing so however requires an atomic bond between such an adatom and the tip, thus cutting the tunnelling current and essentially stopping all feedback. For this reason, a simulation program was created, intended to run in parallel with the experiment: the same input that is sent to the stm device is also received by the simulation, which then provides visual feedback based on a prediction of reality.

In this simulation, the stm tip, which consists of thousands of atomic layers, is represented by a skeleton and a number of flexible atoms attached to it. The skeleton is directly controlled by the input device as if it were a character in a computer game; the other atoms are then dragged along according to the gold-gold interaction forces described in chapter 2. An example of such a tip that is used for moving a gold adatom is shown in figure 3.1. The skeleton can appear in many forms, such as a single layer, a conic structure or even the entire shape of the tip. Which amount of skeleton atoms is appropriate depends on the stiffness of the tip we want to simulate: more skeleton atoms means a stiffer tip; more flexible atoms means a softer one.

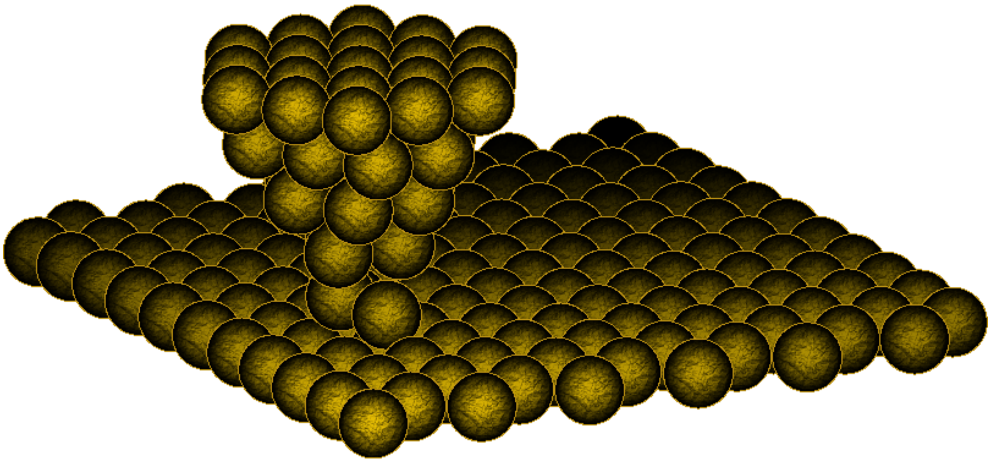


Figure 3.1. *A screenshot of the old simulation program in action. In this situation, the stm tip is used to push around a gold adatom lying on the surface. The bulk of the tip and the piezo elements used for controlling such a tip in reality are abstracted away by the top layer or boundary layer, which is moved with an external input device.*

3.2 Extension to molecular electronics

In the new experiment, for which we are modifying the molecular dynamics simulation, the starring role is played by an organic (either aliphatic or aromatic) molecule, which lies on the gold substrate. For the duration of the project, this place was assumed by 1,4-bis(4-pyridyl)benzene, whose structure formula is shown in figure 3.2; this can however be easily changed based on the end user's needs. The idea is then to lift this molecule from the surface by picking it up with the stm tip at one of the ends. As mentioned in section 2.3, nitrogen tends to form a stronger bond with gold atoms—especially the tip apex atom—than the other atoms in the molecule (carbon and hydrogen), and this is exactly what enables us to do so. In this way, by having the molecule stand vertically on the surface, we can effectively form a molecular bridge between two gold electrodes. Such a bridge then makes up for interesting study material in molecular electronics research.

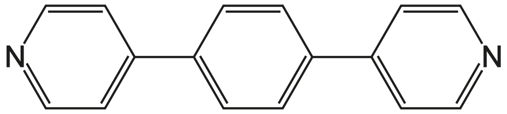


Figure 3.2. *The structural formula of 1,4-bis(4-pyridyl)benzene.*

3.3 Increasing the substrate size

One of the problems with this simulation is that its performance is very poor when the substrate size is increased, allowing for only up to approximately 350 atoms. At a higher number, the program no longer runs smoothly or even freezes for a couple of reasons while doing the necessary calculations. The total time needed to execute 100 molecular dynamics iterations (calculate forces, integrate equations of motion, apply thermostat), which we take as a unit value from now on, is shown in figure 3.3 for different substrate sizes.

Of course, this is not a realistic situation, for the marcoscopic surfaces that are usually studied in these types of experiment consist of uncountably many atoms. These surfaces are usually of a very large area compared to the molecule, and many bcc- or fcc-oriented layers deep. Out of a mere 350 atoms we can only build a substrate a handful of layers deep, with a small area, spanning a few molecules in width and length. At this size, the molecule is affected by deviating motion of the atoms at the boundary. Since these atoms have neighbouring atoms on just one side, they feel less attraction forces and are therefore likely to produce strange effects like moving off the substrate or 'curling up'. In order to increase our working space, we must accelerate the program so that it can process a higher number of atoms in the same amount of time. This, together with the problem described in the next section, is what we hope to solve with general purpose gpu (gpgpu) programming.

3.4 The timescale problem

As mentioned in chapter 2, choosing the right value for the timestep Δt is vital to obtaining proper particle motion. In this sense, it is mostly a matter of striking the right balance between speed and accuracy. A higher timestep results in a faster simulation since less calculations need to be done to reach the same point in simulation time; however, such timesteps produce large displacements and in this way increase the error introduced by discretisation of time. A lower bound to the timestep, on the other hand, is enforced by the requirement that changes should be visible in real-time.

For a system made up purely of gold, a timestep of the order of femtoseconds is sufficient[2, 16]; with the presence of a molecule, this is no longer the case, for it causes instabilities in the molecule. After all, the mass of molecule atoms is more than an order of magnitude smaller than that of gold atoms, while the forces acting on these atoms are larger. When using the same timestep, the calculated displacement will bring the atom to a position

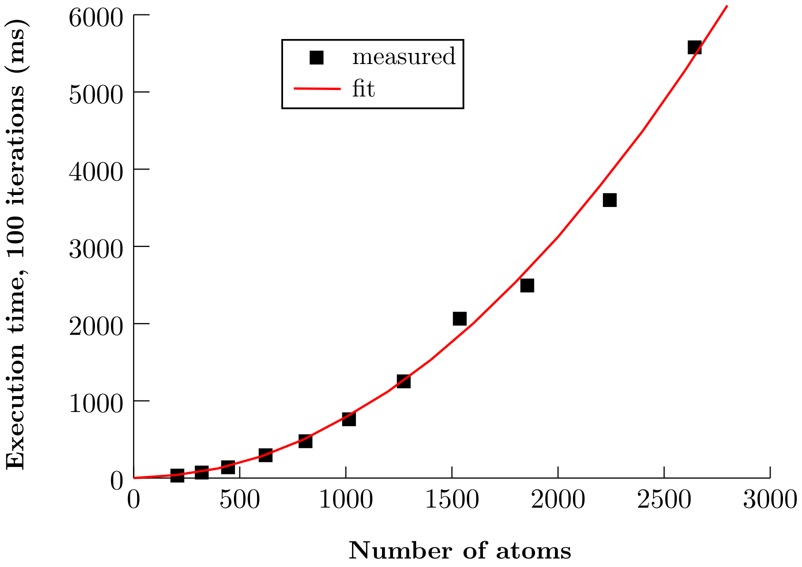


Figure 3.3. *The amount of time it takes the old simulation program, which was fully implemented on the central processing unit (cpu), to finish 100 iterations of gold atom relaxation. Quadratic dependence of this execution time t_{ex} on the number of atoms N (which includes both tip and substrate atoms) is clearly visible. The data points are fitted according to the quadratic function $t_{\mathrm{ex}} = 0.028N^2$.*

$\mathbf{r}(t + \Delta t)$ where its potential is *higher* than the potential at $\mathbf{r}(t)$; since it is subject to a sum of several harmonic oscillators, this effect is amplified every step, which causes the molecule to explode after a short time. This is illustrated with a simple quadratic harmonic oscillator in figure 3.4.

As such, we see ourselves forced to use a timestep that is smaller by more than a magnitude. Apart from slower gold motion in general, this has a noticeable consequence, namely that the tip atoms bound to the skeleton cannot keep up with its movement, which is controlled with the input device. Although one could argue that moving the tip boundary at a lower speed would circumvent this issue, the goal of the original project was to make molecular dynamics simulation interactive and useful as an auxiliary tool for conducting stm experiments. In short, simply scaling down the timestep is not an acceptable way of dealing with the problem, and we must find another solution.

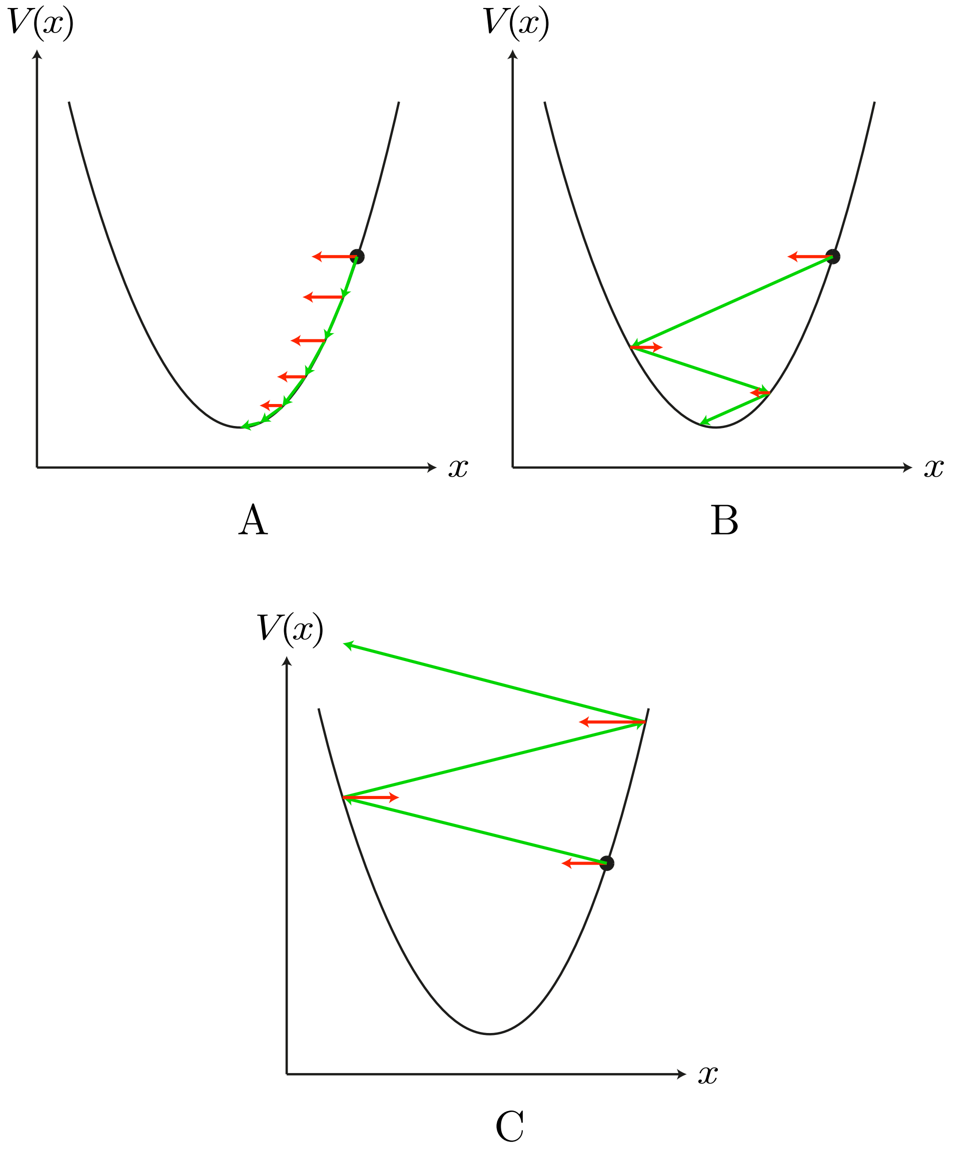


Figure 3.4. *Approximate trajectory of a particle in a simple harmonic potential, for three different values of* Δt*. Displacements after each timestep are marked with green arrows, while the forces at each position are shown with red arrows. The initial force is equal for all three cases. (A) When* Δt *is small, the atom descends the potential well in many small steps, taking a long time to reach the minimal energy position. (B) At a larger timestep, the particle experiences a damped oscillation, arriving at the bottom more quickly. (C) For a too large timestep, the atom ends up in a position with a higher potential energy, causing a snowball effect and eventually launching it into oblivion.*

Chapter 4

Graphics card programming with Cuda

4.1 Motivation

Originally, video cards were invented for the purpose of lifting some of the heavy work that comes with graphics processing from the cpu. What makes these gpus so much better suited for these jobs, then, is that they exploit their *parallellism*, or better to say, ‘parallellisability’, by running relatively small pieces of code on *thousands* of processing cores, as opposed to cpus which currently only have 4 to 32. For example, most graphics processing activities include algorithms for rendering, linear algebra calculus or raycasting, all of which consist of a large number of mutually independent computations, that may be executed simultaneously. It was this observation that led to the introduction of *general purpose* gpu (gpgpu) programming, which generalised this concept for other applications, such as neural networking[17, 18], scientific data processing and atomic simulations.
The feasibility of this approach for implementing molecular dynamics simulations has been shown in numerous projects and experiments[19, 20]. This is not surprising, as the bulk of the execution time is consumed by routines calculating the interatomic forces, which do not in any way depend on one another. After all, these forces are derived from potentials that are functions only of atom positions and time. With that in mind, we have chosen to use this technique in order to accelerate our own molecular dynamics simulation.

4.2 Cuda vs. Opencl

One of the first decisions that one has to make before starting to build a gpgpu implementation is which programming framework to choose. At the current moment, the two most widely deployed apis in this field are the *Compute unified device architecture* (Cuda), which was developed by Nvidia, and the *Open compute language* (Opencl) api, maintained by the Khronos group. While Opencl has a number of advantages over Cuda—most notably the fact that it is aimed at cross-platform development (extending to digital signal processors, field-programmable gate arrays (fpgas) and others), whereas Cuda only runs on gpus manufactured by Nvidia—, recent studies[21] have shown that Opencl is much more verbose, and has a steeper learning curve than Cuda. For us, this is a significant drawback, seeing as the development of this program is expected to be continued by others, who might not be experienced in this field. The requirement for more easily readible and understandable code, therefore, has lead us to choose for the Cuda framework.

4.3 Programming in Cuda

The Cuda programming interface provides an abstraction to gpu programming through the use of a number of models, most importantly the *programming model* and the *memory model*[22]. The programming model drives the programmer to follow a certain structure in writing his code, which is then mapped to the gpu architecture. This allows for optimising code at a low level, while retaining the functionality and high flexibility of the C++ language. The memory model obeys this same principle, requiring to some extent manual memory management—that is, manual memory allocation and deallocation on the gpu, and explicit data copying—without having to interfere with the details of transfers at assembly level.

4.3.1 Programming model

The basic building block of any Cuda application is a *kernel*, which is basically a function running on the gpu that may be called either from the cpu or the gpu. It differs from usual functions in that many instances of the same instructions defined in kernels are executed by what are called *threads*. Threads are chunks of sequential code that run on the gpu processing core on the gpu that they were assigned to. A high level of parallellism, then, comes from having many threads running on many cores at the same time. These threads are grouped in *thread blocks*. For convenience, these blocks can be defined in up to three block dimensions, thus giving each thread an index vector $[\textsc{t}_x, \textsc{t}_y, \textsc{t}_z]$. While the execution of these threads happens in the same way irrespective of whether one uses multiple thread dimensions or not, doing so can ease the design of algorithms that have to handle multi-

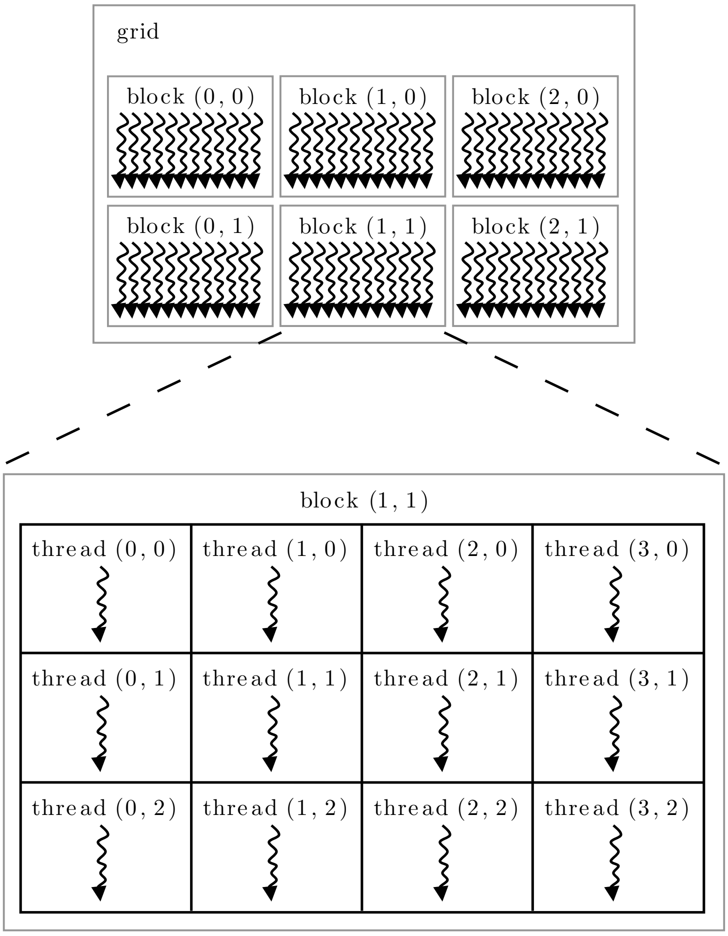


Figure 4.1. *A twodimensional grid of blocks, where each thread block also has two dimensions. Typical thread blocks will in reality contain more than twelve threads, since threads are dispatched in groups of 32 on the gpu.*

dimensional data, such as matrices. In the same fashion, groups of thread blocks are packed together into a *block grid*, which may again have up to three dimensions. An example is shown in figure 4.1[22]. The size of each block grid is normally dictated by the size of the data set to be processed by the kernel, while block size is often algorithm-specific. Depending on the algorithm, it may occur that calling a kernel with a certain block size may result in better performance than other sizes, and the optimal block size may depend on many variables[23]; hence, this size is often chosen empirically.

4.3.2 Memory model

Similar to a cpu having various types of memory—drive memory, random access memory (ram), register memory and others, depending on the system—, memory on the gpu is modelled into certain levels. Data is transferred between cpu and gpu via the pci-e bus, connecting the cpu ram with the global gpu memory, also dubbed device memory. Global memory in turn can be accessed by all threads, through an L2 cache. Within each thread block, threads can write to a chunk of so-called shared block memory, which may vary in size as defined by the user upon calling a kernel. This shared memory is an abstraction of L1 cache equipped memory within each streaming multiprocessor (simply put, a group of processing cores put together with a number of other functional units), which can be dynamically allocated. Since this shared memory is located closer to the cores, and uses a smaller cache, shared memory latency is a lot smaller than global memory latency; as such, it is the fastest route for communication between threads. A lot of performance optimisation involves maximising shared memory usage above global

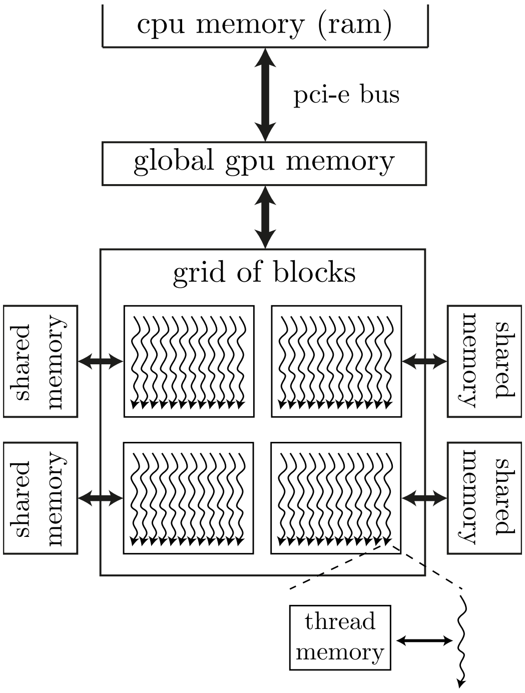


Figure 4.2. *The Cuda memory arrangement. Each grid of blocks (that is, all threads within a grid of thread blocks) is connected to the global memory, which is bridged to the cpu ram via a pci-e bus. Each thread block has a separate chunk of block memory, and each thread can also use its own thread memory.*

memory. One must keep in mind, though, that shared memory is limited, and using too much of it per block will reduce the number of blocks that can run in parallel.

Lastly, each thread can store its local variables in thread memory, which is implemented as a number of 32-bit register files. These are the fastest parts of gpu memory, but cannot be shared with other threads. A schematic view of the Cuda memory model is shown in figure 4.2.

4.4 Plan of action

As mentioned in the first section of this chapter, the benefit of implementing a molecular dynamics simulation on a graphics card lies in the high degree of parallellism that can be achieved when many independent tasks are executed simultaneously. Therefore, our goal should be to select exactly those segments of the program that are most suitable for being parallellised, and redesign the algorithms for these respective tasks. After all, not every component is suitable for running on a gpu: while the graphics chip may contain many more processing cores than the cpu, they run at a lower clock speed and lack an interface for correspondence with any other devices.

Taking all this in consideration, we decided to reimplement on the gpu the following parts:

1 force calculations;

2 velocity Verlet integration;

3 the thermostat (i.e. the constant friction factor mentioned in chapter 2).

The elements remaining on the cpu, then, include handling the data from the input apparatus, moving any recorded force values to a small LabView graphing program for real-time plotting, and controlling the visualisation of the atoms on the screen. One might now be wondering why the latter is assigned to the cpu even though drawing atoms on the screen is clearly a matter of graphics processing. The answer to this question is that we use a graphics library[24] which has been implemented in terms of cpu-callable methods, but in fact lets the bulk of the work be done by the integrated graphics chip (which is different from the card that we use for gpu programming).

Shown in figure 4.8 at the end of this chapter is a flowchart that includes all core tasks performed by the program.

4.4.1 Tackling the timescale problem

In this section we propose a solution to the problem of different timesteps, as we described in section 3.4. Since the amount of operations required to compute the interaction forces between gold atoms in the substrate is much higher than for the other forces (especially in the case of large substrates), any attempt to accelerate these calculations by such a large factor that would completely bridge the gap between the two timesteps—that means having this kernel run at the same speed as the kernels computing the gold-molecule interactions and intramolecular forces, while retaining the speed to run the simulation in real-time—is very likely to fail. Fortunately, it's precisely the gold-gold forces that have proven to work with the large timestep; therefore, we will keep this timestep and focus on the forces involving the molecule instead. In order to overcome the difference in elapsed (simulation) time between the inertial system of the molecule and that of the gold atoms, we repeat the whole procedure of calculating forces, integrating them and applying the thermostat a number of α times, where $\alpha = \Delta t_{\text{gold}}/\Delta t_{\text{molecule}}$. We seek to maximise performance by letting the kernel that is responsible for the substrate interaction forces execute simultaneously with the sequence of α molecule relaxation steps. The tool that the Cuda specification provides to achieve this is the structure of *streams* which can be defined to run in parallel on the gpu.

It is not immediately clear whether this method preserves correctness and the accuracy associated therewith. After all, the motion of the gold atoms, which occurs with a high timescale, carries an error with respect to a molecule system that is updated at a higher frequency and thus with a higher degree of precision. Note that since the molecule exerts a force on nearby substrate and tip atoms, these have to be integrated with the small timestep as well. This leads to a bit of an awkward situation where each gold atom moves multiple tiny distances, followed by a larger leap from the gold-gold interactions. These displacements are however still very small for each iteration—between twentieths and hundredths of Ångströms. Moreover, the mostly convex potentials that the forces were derived from generally create negative feedback and relax the system rather than bringing it to a chaotic state; because of this, small changes and errors usually go unnoticed. It remains an open question though whether this approximation is theoretically justifiable, as the equations of integration become large and confusing very quickly as soon as one tries working out the difference in atom positions between α smaller timesteps and one big timestep. Hence, we will leave the formulae behind for now, and accept a qualitative argument as a basis for our approach.

4.4.2 Key assumptions

In the act of constructing the algorithms as efficiently as the resources of the graphics card and our imagination allow, it turned out necessary to make a number of critical presuppositions. Each of these constrain the amount of atom configurations that the program supports, but were made for efficiency and simplicity purposes.

We assume the following:

1 the molecule in question is *planar*—in other words, all atoms in the molecule, including the carbon atoms, are bonded to at most three other molecule atoms, and any nitrogen atom has a bond with no more than two. This is merely a restriction on the number of atom bonds, and vibrations out of this plane may still occur;

2 the kernels that are responsible for calculating the intramolecular forces can take at most 32 molecule atoms per thread block;

3 the size of the cells that we use in our neighbour cell data structure, as described in section 4.5, is restricted to 32 atoms;

4 covalent bonds are immutable, and cannot be broken, formed or changed over the course of the simulation. Since we are dealing with only one molecule that is not undergoing any chemical reactions, and the intramolecular bond forces are much stronger than the attractive forces between the molecule and the other atoms, we decided to keep these bonds fixed.

The rationale behind these assumptions, in terms of implementation practicality, is explained in the sections describing the algorithms below.

4.4.3 Technical specifications

The machine we use for simulating the experiment is equipped with an Intel i7-3770 cpu, which has 8 logical cores, and 16GB of ram at its disposal. Rendering on the screen is done by an Intel hd graphics 4000 integrated graphics chip. The graphics card that accelerates the program is an Msi GeForce GTX 960, with 4GB of gddr5 device memory. This gpu is of the Nvidia Maxwell architecture, and supports Cuda compute capability 5.2.

4.5 Gold-gold forces

The algorithm that we use for calculating the interaction forces between the gold atoms in the substrate and the tip is to a large extent based on the *cell*

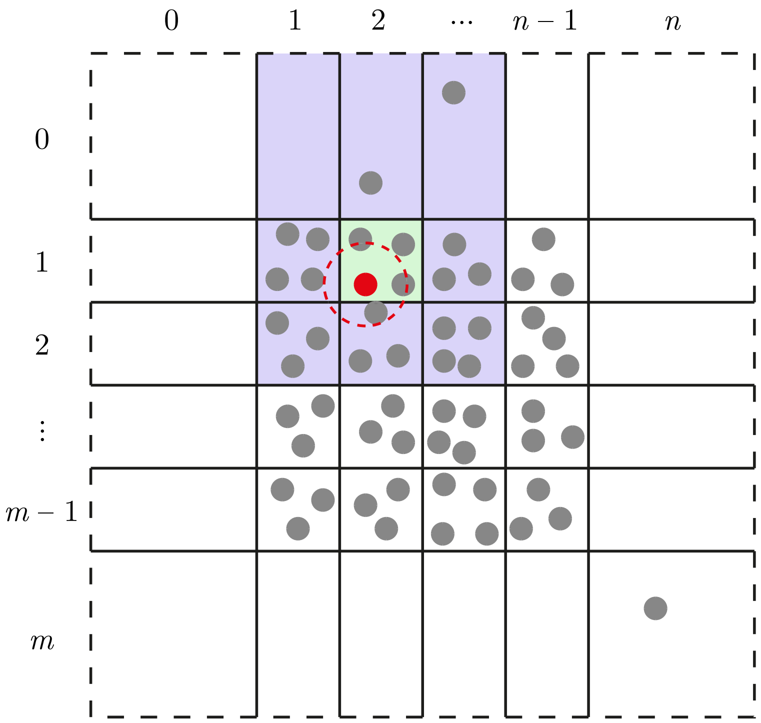


Figure 4.3. *Gold substrate divided in cells. For the atom coloured in red, the cut-off radius is indicated as a red, dashed circle, while its own cell is depicted in green and its neighbouring cells are coloured in blue. Atoms whose positions are outside the cut-off sphere of the indicated atom are not taken into account in calculating the total force on said atom. Note that the outermost cells have no boundary; this measure was taken to fit all of 3D space—an atom may end up anywhere, after all—into a finite number of cells.*

list method[25]. This approach differs from others such as the neighbour list method[26] in that it places each atom in one of many cells that together make up the whole substrate and, in this case, the tip structure. The size of these cells is fixed and depends only on the cut-off radius. Indeed, in our implementation, the dimensions of these cells are set to equal this cut-off distance. This way, we can guarantee that all neighbouring atoms that are located within the cut-off radius of atom i, must lie in one of the cells directly neighbouring the cell that atom i was placed in. See figure 4.3.

This method has several significant advantages:

1 it imposes a data structure on the atom objects according to their position, something which is not practically achievable with a simple atom list;

2 since we know that all atoms interacting with atom i must be in one of the neighbouring cells, only few instead of all distances between atoms now have to be calculated. Furthermore, the worst case amount of distances to be computed for each cell does not depend on the size of the substrate. This fact reduces the complexity of the algorithm from $O(N^2)$ to $O(N)$.

It must be mentioned that we chose to let all cells stretch from negative infinity to positive infinity in the z direction. This was done primarily to ease implementation. However, it constrains the substrate geometry in that the 32-atom cell limit, which is imposed by the size of the shared memory used for each cell, must be satisfied for the simulation to work properly—any atoms exceeding this limit are not taking into account in the force calculation. Because of this, the substrate may consist of only up to about five layers, depending on the cut-off radius.

Setting up and maintaining this cell structure over the course of the simulation requires another few steps. First of all, the cell index of each atom, a function of position, must be determined. Then, in order for the kernel that will compute the forces to find the atoms belonging to a particular cell, the atom list must be sorted, using the cell indices as sorting keys. As developing our own parallel sorting algorithm falls out of this project's scope, we use the function `sort_by_key()` from the Thrust library[27] to do this. However, it also needs to know how many atoms are present in any given cell. A new list of cell counts is filled by a function which determines these values from the cell indices list. This is depicted in figure 4.4.

It goes without saying that all the functions accounting for this data structure management can run for all atoms in parallel. This ensures that the caused overhead stays small relative to the total force computation time. However, it appeared that refreshing the cell index and cell count lists only once every 20 iterations had barely any effect on the accuracy of the simulation; thus, we chose not to call this routine every iteration.

What makes the cell list approach even better is the fact that it offers a nice way to make use of thread blocks and, more importantly, shared block memory. When handing over each atom cell to one thread block, which runs one thread for each atom that executes a loop over all neighbouring atoms, we can see a lot of data reuse occurring. Indeed, the positions of all atoms in the neighbouring cells must be known to each thread. Following this observation, it makes sense to have the threads in each block first load all of these positions into shared memory, before going on to check whether the distance falls within the cut-off radius and eventually calculating the force. Note that this would not have worked out without the cell structure, since the atom positions of the whole substrate would require such a large chunk of

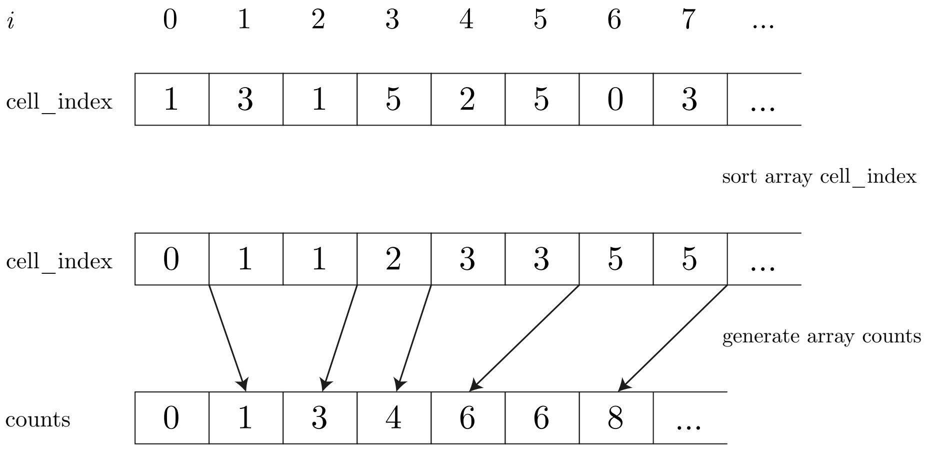


Figure 4.4. *Procedure for setting up the cell data structure. For every 'jump' in the cell index list—that is, every pair of successive entries that have different values—the number of atoms in this cell plus those in the previous cells is inserted at that index in the cell counts list.*

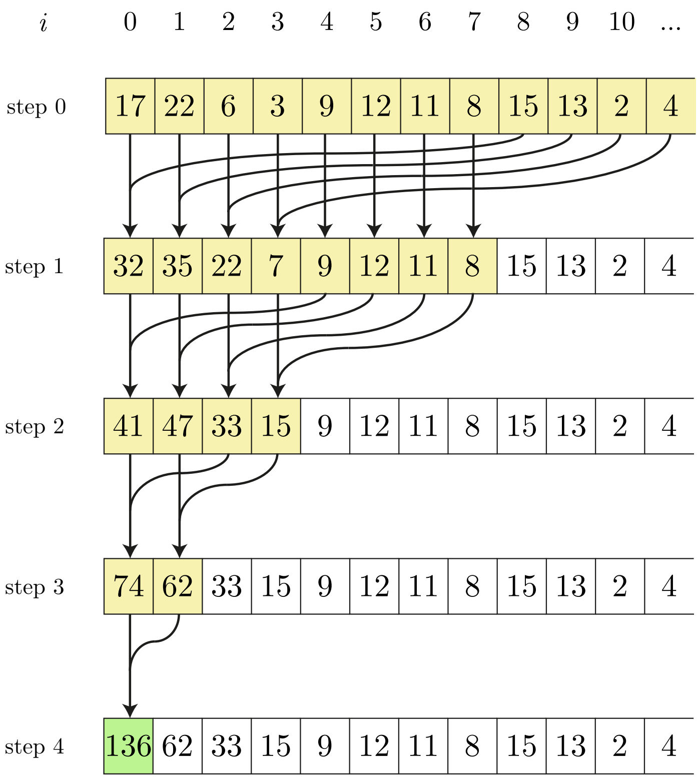
shared memory that it would severely limit the number of blocks that could run on the gpu at the same time.

4.6 Gold-molecule forces

For computing the interaction forces between the molecule atoms and the gold atoms, roughly the same principle is followed. However, keeping the molecule atoms and gold atoms in one list, thus losing the consecutiveness of the molecule atoms through sorting by cell indices, is impractical for a number of reasons. First of all, having to find out which of the atoms belong to the molecule every time before calculating the intramolecular forces would hurt performance and readability. Secondly, one may want to perform some processing on the cpu, such as graphing, and having two separated lists for the gold part and the molecule makes that considerably easier. This choice introduces the need for another cell index and cell count list; this is not a problem though, because the size of the working space, which is the number of atoms, does not change.

Unfortunately, we found no obvious way to use Newton's third law of action and reaction. The issue with this law is that it is very likely to produce many data race conditions, since the force applied to one atom is a sum of many force terms, one for each atom with which it interacts. Data races are a common problem in parallel programming, and happen when two threads try to change some value in the same memory address: both read and change the value at the same time, which causes the returned value of the fastest thread to be overwritten by the other thread, thus giving the wrong result. Because of this, we need to employ two different kernels, where one computes the force applied to the gold atoms, and the other the force applied to the molecule atoms. In order to decide, then, for any molecule or gold atom, which atoms from the other structure are close enough to be taken into account for the force computation, the function must look at the other cell index and count list.

Another difference in these kernels as compared to the gold-gold force kernel is the use of a second thread dimension instead of a single **for** loop.



i	0	1	2	3	4	5	6	7	8	9	10	...
step 0	17	22	6	3	9	12	11	8	15	13	2	4
step 1	32	35	22	7	9	12	11	8	15	13	2	4
step 2	41	47	33	15	9	12	11	8	15	13	2	4
step 3	74	62	33	15	9	12	11	8	15	13	2	4
step 4	136	62	33	15	9	12	11	8	15	13	2	4

Figure 4.5. *Parallel data reduction. The idea here is to sequentially add pairs of variables that reside in shared memory, thus creating a 'waterfall' of additions that ends up as a sum stored in one value. Explicit thread synchronisation is required between addition steps. Intermediate values are shown in yellow, and the end result is coloured in green.*

This was done as we noticed that there was a lot of room for improvement in terms of occupancy. Adding another thread dimension means launching blocks of 32 × 27 threads instead of simply 32 threads. Why 27? Because it is the largest multiple of 9 (the number of cells that need to be considered for each atom) smaller than 32, thus bringing the amount of threads per block as close as possible to the limit of the Maxwell architecture, which is $1024 = 32 \times 32$. Each of these 27 threads still executes a (much shorter) **for** loop in order to cover all atoms in its own cell. One might be wondering, then, how we deal with the problem of race conditions that this method certainly causes. However, the fact that these data races happen only within the block makes them much more manageable, and enables us to use (a one-block version of) the so-called *parallel reduction method*[28]. This method is described in figure 4.5.

4.7 Intramolecular interactions

The last part of the force calculation consists of intramolecular interaction forces, which in turn are divided into bond forces, angular forces and torsional forces, as detailed in chapter 2. Since these interactions in reality happen between atoms that share a covalent bond, whose length may vary over time, we cannot use the cell list to model them; another data structure is necessary. In our case, all molecule atoms have up to three references (which are not references in the strict sence of C++, but merely index numbers) to another molecule atom. An additional kernel redefines these references after sorting the atoms for the cell list, to ensure that they point to the correct atoms after their index in the list has changed. In this way, we can treat the molecule as an undirected graph.

The force on each atom is then computed by finding all bond pairs, angular groups and dihedral groups which the atom is part of, and summing up the force contributions from each of these groups. Again, we use a second thread dimension in order to maximise occupancy, and apply parallel reduction to obtain the correct sum. We pick the size of the thread y dimension to be the theoretical maximum amount of these groups, and each thread y coordinate T_y is mapped to its respective group. Since the maximum amount of bond pairs plus the maximum number of angular groups is less than 32, we merge these two routines into one kernel, so as to reduce kernel launch and stream synchronisation overhead.

4.7.1 Bond and angular forces

Since the bond forces and the angular forces are computed by the same kernel, it is necessary to separate the threads according to the task they need to perform. In our implementation, the first three threads in the y dimension ($\mathrm{T}_y = 0, 1, 2$) are assigned the duty of calculating the bond forces, while the rest is set to work out the angular forces. Whereas the routine for the bonds is very straightforward—simply look at all neighbours and calculate a spring-like force—, resolving the angular forces is a bit more delicate. This stems from the fact that for each group we need to consider multiple different atoms, some of which are not direct neighbours of the atom in question. On top of that, we must distinguish between two types of angular groups:

I inner groups are those triples where atom i, for which we are calculating the force contribution, is the middle atom;

II outer groups are the angular groups that have atom i at one of the ends.

An example of each group is depicted in figure 4.6. The maximum amount of angular groups in a planar molecule is 9; this follows from the observation that for a maximum of n bonds, the number of inner bonds is at most $\frac{1}{2}n(n-1)$, while the number of outer bonds is capped at $n(n-1)$, giving a total of 9 for $n = 3$. The atom indices j and k, as shown in figure 4.6, are then

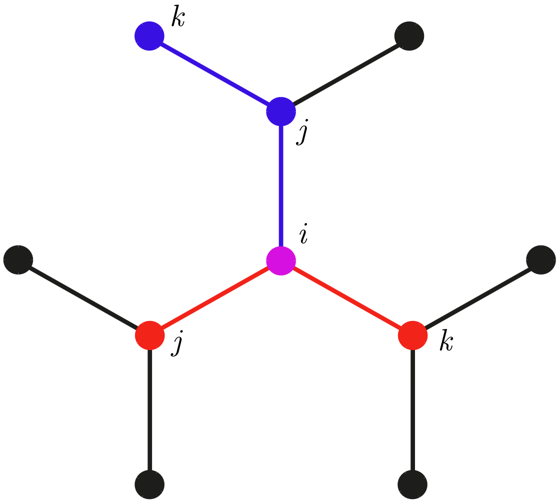


Figure 4.6. *A planar atom structure whose atoms all have to be taken into consideration in order to find which angular groups an atom i interacts with. An inner angular group is shown in red, and one outer group is displayed in blue. Atom i is also a member of both groups; hence, it appears in purple.*

obtained by traversing the graph. Index j may be picked as one of the direct neighbours of atom i; for inner groups, k is selected to be one of atom i's neighbour which is unequal to j; for outer groups, we choose k as one of atom j's neighbours which is different from atom i. Each pair of indices (j, k) can be chosen uniquely for every τ_y between 3 and 11. Once the angle groups are found, we apply the force derived in chapter 2 to atom i.

4.7.2 Torsional forces

The procedure for finding the torsional groups is similar to finding the angle groups, with the only distinction being that we now have to deal with groups of four atoms instead of three. Fortunately, we still need to consider only two types of torsional groups, just like the case of the angle groups:

I in the inner groups, atom i is one of the two middle atoms in the chain of four;

II for the outer torsional groups, atom i is at one of the ends and as such has only one connection to another atom in the same group.

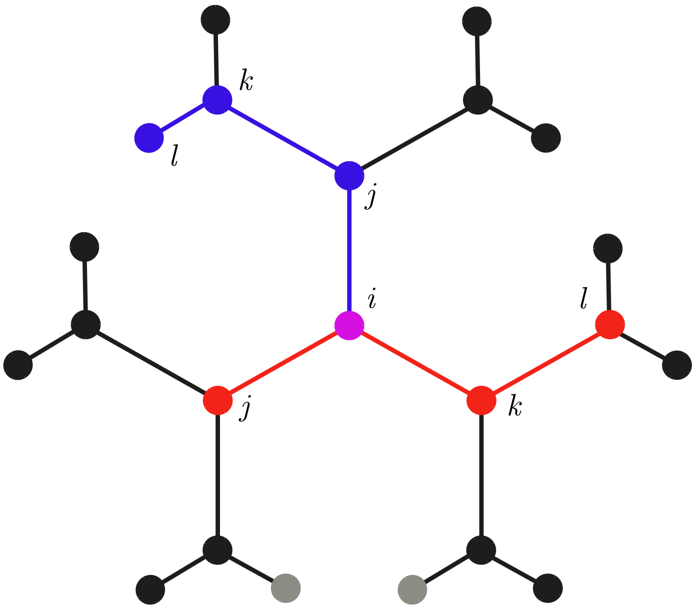


Figure 4.7. *A schematic planar atom structure with torsional groups. An instance of an inner group is shown in red, and an outer group is indicated in blue. Note that pairs of atoms like the two coloured grey may happen to be one and the same atom; this occurs very often in hexagonal aromatic structures.*

The presence of this extra atom requires us to traverse the graph one more level in order to complete the group. Determining indices j and k happens in exactly the same fashion as for the angular groups; indices l are picked as one of atom k's neighbours that are not a reference to either atom i (inner groups) or atom j (outer groups). See figure 4.7.

One can understand now why constraining the number of bonds per atom to three is so important. Again letting n be the maximum number of bonds per atom, we have n options for picking index j and $n - 1$ choices for both k and l, in the case of the inner groups as well as the outer groups. This leads to a maximum number of groups $G_{\max} = 2n(n-1)^2$. Taking $n = 3$, we find $G_{\max} = 24$, a number nicely smaller than 32. However, a change to $n = 4$ increases this amount to 72, which is problematic. For one, we would need to launch the kernel with thrice the number of thread blocks; secondly, since many atoms in the molecule are likely to be hydrogen atoms with only one bond (thus only allowing for four groups), we may end up wasting a lot of threads. In short, the drop in efficiency renders this method much less suitable for n larger than 3. One could argue that this may be solved by precalculating the groups and having the kernels load them from the memory; however, this would require redefining all groups after sorting, and would also sternly increase global memory traffic.

The last thing that should be mentioned is the following. Throughout this project, we have used 1,4-bis(4-pyridyl)benzene as our organic molecule, which consists of 30 atoms. Because of this, one thread block was sufficient for the kernels that compute the intramolecular forces. However, a new experiment might create a demand for a larger molecule, thus requiring more thread blocks to take care of certain parts of the atoms. While this does not change the total amount of work done per thread block, it does increase the total shared memory allocated per block. After all, the data of all molecule atoms needs to be known to each thread block, since the atom indices are changed every time the cell list is updated which means it is not obvious beforehand which atoms will and which will not occur in the bond groups, angular groups or dihedral groups. As such, n times more atoms will require n^2 more bytes of shared memory to be allocated. For very large molecules, this could have a noticable impact on performance, and efficient pruning may be needed to address this problem.

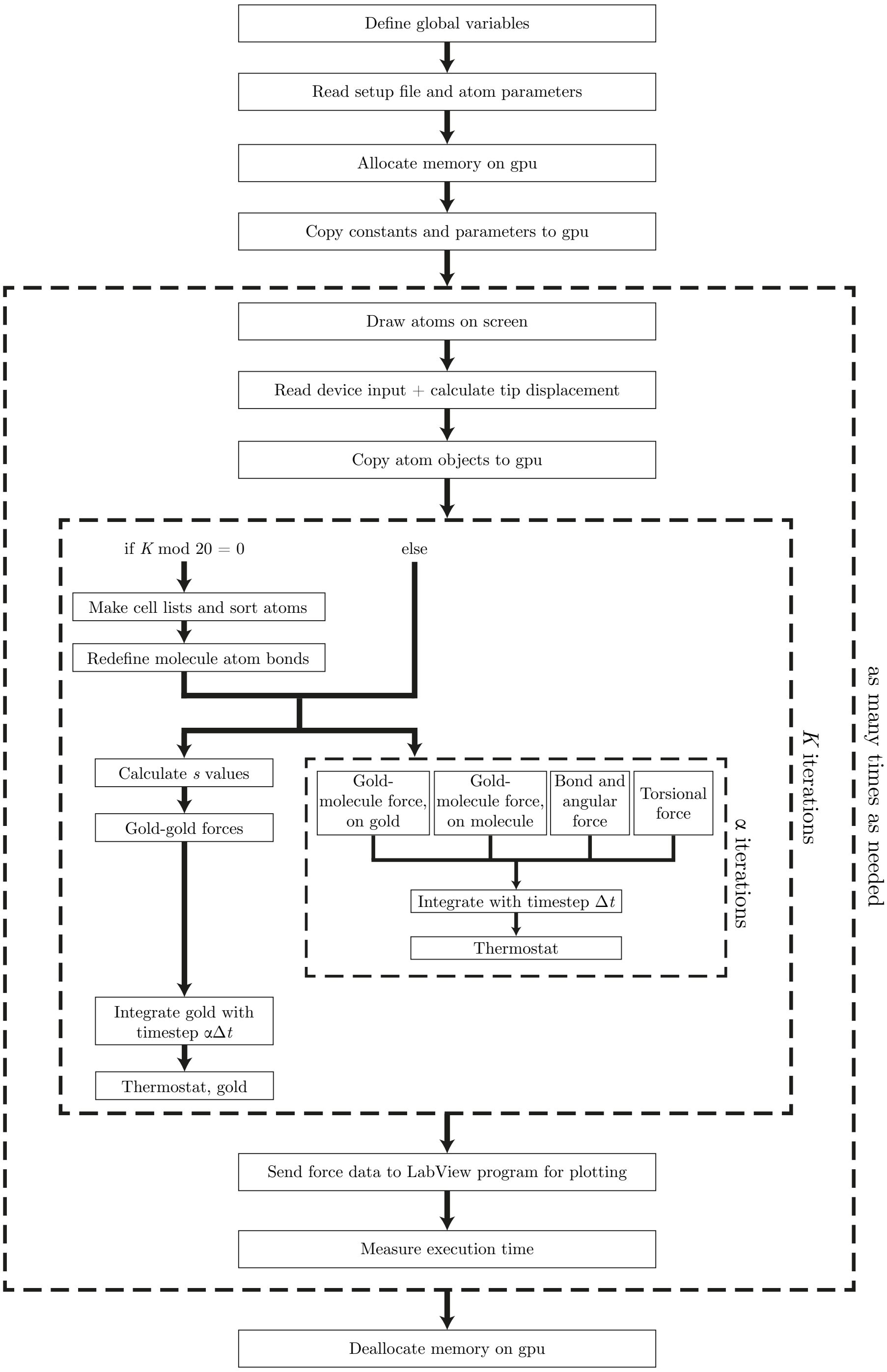

Define global variables
Read setup file and atom parameters
Allocate memory on gpu
Copy constants and parameters to gpu
Draw atoms on screen
Read device input + calculate tip displacement
Copy atom objects to gpu
if K mod 20 = 0
else
Make cell lists and sort atoms
Redefine molecule atom bonds
Calculate s values
Gold-gold forces
Gold-molecule force, on gold
Gold-molecule force, on molecule
Bond and angular force
Torsional force
Integrate with timestep Δt
Thermostat
α iterations
Integrate gold with timestep $\alpha\Delta t$
Thermostat, gold
K iterations
as many times as needed
Send force data to LabView program for plotting
Measure execution time
Deallocate memory on gpu

Figure 4.8. *Flowchart showing the execution scheme of the program. Time runs in the vertical direction, thus subroutines placed next to eachother run in parallel. Parts of the chart contained in dashed boxes run as many times as stated to the right of said box.*

Chapter 5

Results and discussion

In chapter 3 we set out our objectives for this project, which to an extent boil down to one common goal: accelerating the simulation program. In this regard, we begin by presenting and discussing the results of our performance measurements, together with several general notes on gpu programming in this light. Subsequently, we discuss the accuracy of our gold-molecule interaction model, in order to determine its feasibility as a sufficient description of our experiment.

First of all, we want to make a performance comparison between the cpu functions that compute the gold interaction (including integration of motion and velocity rescaling with a friction factor thermostat) and the corresponding kernels that run on the gpu. This should give a solid idea of how well gpgpu programming methods can be used for increasing the size of the substrate to which we apply the simulation. The results, in the form of total time that is needed to execute one hundred of these iterations on both cpu and gpu, are summarised in figure 5.1, which also includes the graph in figure 3.3. We can immediately see a striking difference in performance between the two: where the execution time of the cpu implementation quickly reaches the order of seconds, the gpu stays under 100 milliseconds even at a much higher number of atoms. What is interesting is that the linear dependence of the execution time on the number of atoms, which is inherent to the use of a cell list data structure, only appears to hold from about 2000 atoms on. A possible explanation for this is that the gpu is not yet completely filled before the next synchronisation point (which is the end of an iteration), and could take on more threads. In short, it seems that the gpu is very well suited for this task. Indeed, it runs one kernel spread over many threads to calculate the forces for all atoms, and thus exploits the parallelism of the system.

The gold-molecule interactions, together with the intramolecular forces,

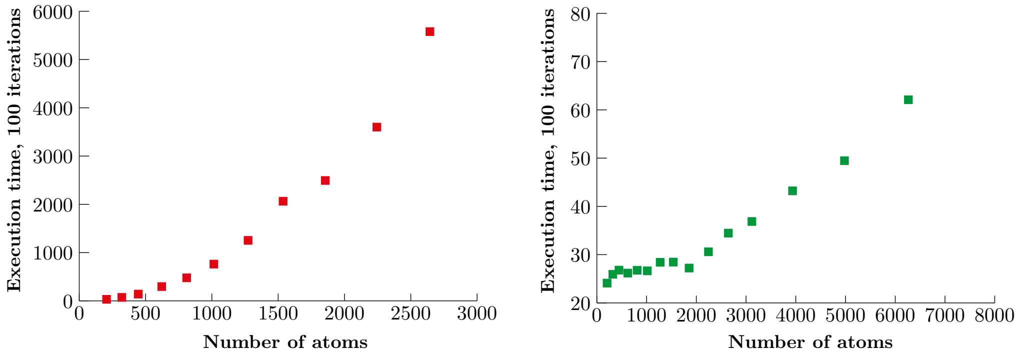


Figure 5.1. *Total execution time, in milliseconds, for 100 iterations of atom relaxation with only gold interaction and motion enabled. The left graph shows the performance of the old cpu simulation, while the calculation speed of our gpu simulation is displayed in the right graph. In addition to the notable difference between the two implementations, we see that the time it takes the gpu to execute 100 iterations is nearly constant for a small number (less than 2000) atoms, while a linear relation is observed in the cases with a higher number of atoms.*

are however a completely different story. Our investigation of this matter was done with the smallest setup that was used for measuring the performance of the two implementations, which contains 206 atoms (including the tip and the molecule). Running the small timestep loop α times together with the larger timestep sequence, as explained in section 4.4.1, increased the total execution time to around 82 milliseconds per 100 iterations. For testing purposes, we also considered a serialised smaller timestep loop; this approximately doubled the run time. What we can learn from this is that parallelising the small timestep loop using streams does net a performance boost, which however turns out to be smaller than we hoped given the amount of tasks that are expected to be carried out simultaneously. We believe that the gpu getting filled up is the primary reason for this, causing the rest of the operations to be stalled. Another source of delay, revealed by a profiling tool that is part of the Cuda development kit, is the kernel launching overhead. Since the threads operate on little data and thus finish their job quickly, this overhead becomes relatively large and can even dominate the execution scheme in terms of latency.

In order to overcome this issue, one could suggest moving the computation of the intramolecular interaction forces back to the cpu, since the cpu cores have a much higher clock frequence and are therefore likely to execute these seemingly sequential operations quicker than the gpu. There are however several problems with this. For one, constantly sending back and forth the molecule data (that is, the positions and possibly velocities and forces of the molecule atoms) would introduce significant memory latency over the pci-e bus, especially since the size of the data is so small that the peak bandwidth is never reached. Secondly, one iteration of intramolecular relaxation is computed so fast that synchronisation between the cpu and the gpu becomes an issue. In the current Cuda implementation, concurrently running gpu kernels can be efficiently synchronised (using Cuda *events*), while explicit cpu-gpu synchronisation comes with a large delay, which effectively more than doubles the execution time.

Another alternative to circumvent this shortcoming of the gpu could be the use of *field programmable gate arrays* (fpgas), which have been proven to be fruitful for accelerating molecular dynamics simulations, among other applications[29]. An fpga, simply put, consists of a large number of functional units, logic gates and connecting parts, all of which can be very precisely configured to build an application-specific processor. Here lies the main difference between fpgas and gpus: gpus are programmed by writing snd compiling software, while fpgas need to be manually configured using a hardware description language. A big advantage of fpgas is that while they can perform many tasks in parallel just like gpus, they are not single-instruction, multiple-data (simd) machines and as such do not inherently suffer from a high degree of branching or serialisation. That said, data transfer between a cpu and an fpga is generally less flexible, and fpga programming (including optimisations) usually takes way more time and effort than gpu programming. Even though higher level interfaces, such as the Opencl C/C++ api, are available, configurations constructed from such code tend to run at suboptimal performance. Given the size of this project, we therefore deemed the fpga method unsuitable for our needs.

All in all, while acceleration by parallellisation, applied through gpgpu programming, shows to be very promising, it must be noted that the complicacy of such implementations quickly increases as we make relatively small changes to the system that we want to simulate. Whereas in a sequential cpu program adding new potentials and forces requires writing only a few more loops and functions, we see that in the gpu case it was necessary to revamp our algorithms and reconsider the required data structures. In other words, parallel programming requires a new way of thinking, and in some cases severly limits the programmer.

Naturally, our discussion would be incomplete without any comments on the physical accuracy of the new simulation, especially the assumption we made for solving the multiple-timestep problem and the validity of the potentials that we have chosen to model the system. First off, it appears that the atomic structure reacts to the presence of the molecule in quite the same way as the old situation, albeit much faster. It is important to note that the system remains *stable*, despite the errors that were introduced by keeping the gold atoms fixed while the small timestep interactions are being computed. This suggests that these errors are small enough not to perceptibly disturb the ensemble, and thus may be neglected, something which is facilitated by the constant damping from the thermostat. We however only draw this conclusion from looking at the evolution of the visualised system; a better way would be to check observables such as the extent to which energy is conserved[30], and how much this is affected by a multiple-timestep approach[31].

Another matter is the process of individual molecule-gold bond breaking that is often observed in related experiments, and is responsible for jumps that can be seen in the corresponding conductance traces[32]. In our simulation, this is not visible at all. Instead, the molecule slides smoothly off the surface, as if it were a string being picked up from a table. This is illustrated in figure 5.2. Additionally, we noticed some peculiar behaviour after switching off an artificial harmonic potential that we use to keep the tip atoms together. Without this potential, the lowermost of these atoms are pulled off

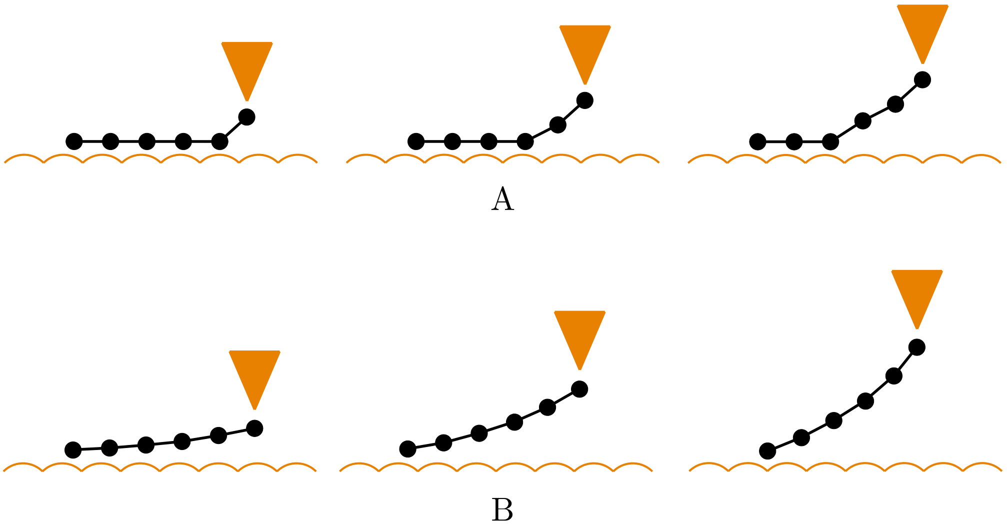


Figure 5.2. *Approximate reaction of the molecule (here drawn as a 1D chain of atoms for simplicity) to the pulling force of the tip, in the expected scenario and our observations. (A) In reality, we peel the molecule off the substrate atom by atom, and we witness the breaking of each bond between a molecule atom and the substrate. (B) In our simulation, the intramolecular forces appear to be much stronger than the gold-molecule forces, creating a very stiff molecule that comes off as a whole instead of sticking to the substrate.*

the tip structure by the strong force derived from the Morse potential, which is clearly not a correct imitation of reality.

Lastly, we felt the need to incorporate the coordination number in the force calculations as it is a good explanation of increased reactivity of the tip apex atom with respect to the other tip atoms and the substrate[33]. In our case however, the $1/n$ dependency of the interaction strength between tip and molecule, where n is the number of direct neighbours, was primarily a measure taken to prevent the molecule from climbing up the tip. Yet, the quantity n is ill-defined when forces are determined solely by interatomic distances, which is the case for all forces that involve gold atoms in some way. Therefore, we feel that a better definiton of this coordination number, together with a reconsideration of the gold-molecule interaction potentials, is necessary to mimic the atomic mechanics at this level in a more reliable fashion.

Chapter 6

Conclusion

Overall, we can say that the application of gpgpu programming with Cuda in order to accelerate the molecular dynamics simulation has been a success. Most importantly, it is the gold-gold force computations that benefit the most from this method, yielding a large speedup relative to the old simulation. For large substrates, we find a relative speedup of up to two orders of magnitude. The presence of the molecule, and the need for executing a greater number of small timestep iterations in particular, somewhat attenuates this gain; the simulation remains easily controllable though, be it at a lower refresh rate (approximately 12 times per second) than what is normally considered to be smooth (24 times per second). The high degree of serialisation is mainly responsible for this.

Our proposal for bridging the two different timesteps required to run the program efficiently seems to work out well in terms of accuracy at first glance. We can however only conclude this from qualitative observations, and thorough investigation of the quantitative physical consequences and the underlying theory are necessary to make a well-founded judgment. As for the rest, it is clear that work still needs to be done to give a more correct picture of realistic motion of an organic molecule that is being pulled from a gold substrate by an stm tip. It is especially the gold-molecule interaction potentials that would benefit from a revision, taking into account coordination numbers in a more decent way.

While we have not been able to finish the extension of the simulation to a much more complex system, it goes without saying that we have made good progress towards completion of the project. Moveover, we have demonstrated the advantages and applicability of gpgpu programming in this context of real-time molecular dynamics, and we wholeheartedly recommend anyone developing their own simulation program to at least take it into consideration.

Chapter 7

Outlook

In this last chapter, we would like to make a number of suggestions regarding further directions that could be explored.

On the background of electronics at nanoscale, graphene has fascinated many a scientist ever since the discovery of its conductive properties. As such, it would be very interesting to see how well we can modify the program to support a graphite substrate instead of a gold substrate, and to what extent this can contribute to the molecular electronics research in a broader perspective. Creating a stable graphite substrate may not appear to be as straightforward as with the gold substrate, since graphene is arranged in a hexagonal structure held together by covalent bonds. Of course, one could exploit the planar structure of graphene to implement a number of approximations, but care should be taken not to oversimplify matters.

Secondly, one could take a closer look at thermostats so that the friction factor, which has been defined rather arbirarily for now, can be replaced. This would enable the reproduction of a canonical ensemble, as initially intended and described in Jacob Bakerman's thesis. One should be wary though that this thermostat could cause a signnificant slowdown if not implemented properly, for the following reasons. For one, this thermostat will have to be applied every iteration, including the small timestep ones. Also, since the thermostat and the temperature influence the whole system, the thermostat function must depend on all present atoms. Certainly, the severity depends on the thermostat that is chosen, and perhaps a simple one could to do the job for a small cost in efficiency; this is of course up to the next person who is interested in continuing the project.

References

[1] A. Aviram and M. A. Ratner, *Molecular rectifiers*, Chemical physics letters **29**, 277 (1974).

[2] J. J. W. Bakermans, C. Wagner, S. Tewari, and J. van Ruitenbeek, *A novel way to control stm-based manipulation : motion tracking and rapid molecular dynamics simulation*, Bachelor's thesis, Leiden university, 2014.

[3] B. J. Alder and T. E. Wainwright, *Studies in molecular dynamics. I. General method*, The journal of chemical physics **31**, 459 (1959).

[4] P. Hohenberg and W. Kohn, *The inhomogeneous electron gas*, Phys. rev. **136**, B864 (1964).

[5] W. Kohn and L. J. Sham, *Self-consistent equations including exchange and correlation effects*, Physical review letters **140** (1965).

[6] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson, *A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters*, The journal of chemical physics **76**, 637 (1982).

[7] L. V. Woodcock, *Isothermal molecular dynamics calculations for liquid salts*, Chemical Physics Letters **10**, 257 (1971).

[8] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, a. DiNola, and J. R. Haak, *Molecular dynamics with coupling to an external bath*, The Journal of Chemical Physics **81**, 3684 (1984).

[9] R. Cortes-Huerto, T. Sondon, and A. Saúl, *Role of temperature in the formation and growth of gold monoatomic chains: a molecular dynamics study*, Physical review B **88**, 235438 (2013).

[10] D. Tomanek, A. A. Aligia, and C. A. Balseiro, *Calculation of elastic strain and electronic effects on surface segregation*, Physical review B **32**, 5051 (1985).

[11] F. Cleri and V. Rosato, *Tight-binding potentials for transition metals and alloys*, Physical review B **48**, 22 (1993).

[12] 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 **117**, 5179 (1995).

[13] A. Markvoort, K. Pieterse, M. Steijaert, P. Spijker, and P. Hilbers, *PumMa molecular dynamics code*, 2005.

[14] G. Binnig and H. Rohrer, *Scanning tunneling microscopy*, Surface science **126**, 236 (1982).

[15] J. a. Stroscio and D. M. Eigler, *Atomic and molecular manipulation with the scanning tunneling microscope.*, Science **254**, 1319 (1991).

[16] F. Jensen, *Introduction to computational chemistry*, 2007.

[17] H. J. H. Jang, A. P. A. Park, and K. J. K. Jung, *Neural network umplementation using Cuda and OpenMP*, Digital image computing techniques and applications , 155 (2008).

[18] V. K. Pallipuram, M. Bhuiyan, and M. C. Smith, *A comparative study of GPU programming models and architectures using neural networks*, volume 61, 2011.

[19] J. van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, and R. Belleman, *Harvesting graphics power for MD simulations*, Molecular simulation **34**, 259 (2008).

[20] J. Anderson, C. Lorenz, and A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units*, Journal of computational physics **227**, 5342 (2008).

[21] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, *Productivity of GPUs under different programming paradigms*, Concurrency computation practice and experience **22**, 685 (2010).

[22] Nvidia, *Cuda C programming guide*, Programming guides , 227 (2014).

[23] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, *Demystifying GPU microarchitecture through microbenchmarking*, (2010).

[24] L. Gomila, *Simple and fast multimedia library*, 2007.

[25] Z. Yao, J. S. Wang, G. R. Liu, and M. Cheng, *Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method*, Computer physics communications **161**, 27 (2004).

[26] A. A. Chialvo and P. G. Debenedetti, *On the use of the Verlet neighbor list in molecular dynamics*, Computer physics communications **60**, 215 (1990).

[27] J. Hoberock and N. Bell, *Thrust: a parallel algorithms library*, 2009.

[28] M. Harris, *Optimizing parallel reduction in Cuda*, Nvidia developer technology (2008).

[29] Y. Gu, M. Herbordt, R. Giles, W. Qin, and S. Vajda, *Fpga acceleration of molecular dynamics simulations*, PhD thesis, Boston university, 2008.

[30] M. E. Tuckerman and B. J. Berne, *Molecular dynamics algorithm for multiple time scales: systems with disparate masses*, Journal of chemical physics **94**, 1465 (1991).

[31] O. Teleman and B. Jönsson, *Vectorizing a general purpose molecular dynamics simulation program*, J. comp. chem. **7**, 58 (1986).

[32] M. Frei, S. V. Aradhya, M. Koentopp, M. S. Hybertsen, and L. Venkataraman, *Mechanics and chemistry: single molecule bond rupture forces correlate with molecular backbone structure*, Nano letters **11**, 1518 (2011).

[33] R. Z. Huang, V. S. Stepanyuk, and J. Kirschner, *Tip-induced atom extraction: effect of tip geometry and its composition*, New journal of physics **10** (2008).

[34] W. Streett, D. Tildesley, and G. Saville, *Multiple time-step methods in molecular dynamics*, Molecular physics **35**, 639 (1978).

Acknowledgments

First of all, I would like to thank Sumit Tewari for helping me get started with this project, guiding me through it, and for the valuable discussions we had. Secondly, a thank you goes out to Jan van Ruitenbeek for giving me an oppurtunity to do my bachelor research project in his physics group, as well as supervising me, together with Fons Verbeek. Lastly, my gratitude goes out to thank Kim Akius for helping me with setting up the hardware, and of course all the group members, whose presence has definitely made this project a joyful experience for me.