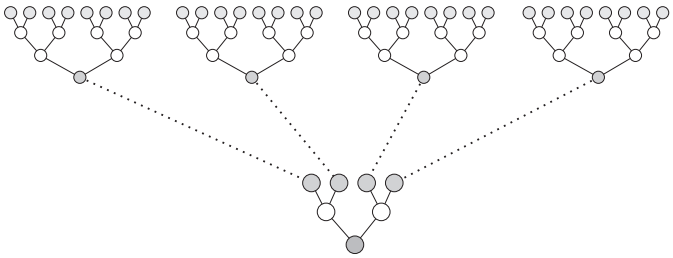
Freie Universität Berlin
Fachbereich für Mathematik und Informatik
Institut für Informatik

Bachelor Thesis Bioinformatics

Parallelization Strategies for a Brownian dynamics algorithm



Ilkay Sakalli

08/08/2008

First Supervisor

Dr. Frank Noé
Bio Computing Group
Free University of Berlin

Second Supervisor

Prof. Dr. Christof Schütte
Scientific Computing
Bio Computing Group
Free University of Berlin

Sakalli, Ilkay:
Parallelization Strategies for a Brownian dynamics algorithm
Bachelor Thesis Bioinformatics
Free University of Berlin

I Ilkay Sakalli declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Abstract

In this thesis we modify, evaluate and implement algorithms which perform in a massive parallel way to boost up Brownian dynamics (BD) simulations. BD-simulations calculate association rates between diffusion controlled molecules. We formulate the sum-product problem and show how to speed up calculations for this problem by using parallelization techniques. A speed up of ≈ 30 is shown to be possible for molecules with huge atom-sizes. Another boost with a factor of ≥ 80 is reached for force (resp. torque, energy) calculations if multiple BD-simulations (≥ 100 BD-simulations) run concurrently. A parallel trilinear interpolation kernel is implemented and evaluated for preprocessing purposes and it is shown how to design algorithms with a performance which will scale up with future GPU devices.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank Mr. Martin Held who helped me correcting mistakes and pointed out important parts and topics to consider. He gave me substantial feedback, helpful advices and suggestions in order to improve this thesis.
In addition, I want to thank Dr. Frank Noé who gave me the possibility to work on a topic which I appreciate very much and which is a pleasure for me to learn and to improve my knowledge in for working in the area of molecular simulations in the future. I also want to thank Prof. Dr. Christof Schütte who agreed to review this thesis as a second supervisor.
It is also an honour for me to thank my family who always listened to me with patience and gave me the energy to work on this thesis although other things had to be done in parallel. I want to thank my little brother to whom this work is dedicated to.

Ilkay Sakalli
Berlin, 08.08.2008

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

SDA	Simulation Of Diffusional Encounter
BD	Brownian Dynamics
MD	Molecular Dynamics
N	Newton
C	Coulomb
SI	Système International d'unités
LE	Charge unit
esE	Electrostatic unit
ϵ_0	dielectric constant
el	electrostatic
PBE	Poisson-Boltzmann equation
APBS	Adaptive Poisson-Boltzmann Solver
UHBD	University of Houston Brownian Dynamics program
GFRD	Green's Function Reaction Dynamics
GPU	Graphics processing unit
GFLOPS	10^9 Giga Floating Point Operations Per Second
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUDPP	CUDA Data Parallel Primitives Library
CUFFT	CUDA Fast Fourier Transform
HPC	High Performance Computing
SPE	Synergistic Processor Element

Chapter 1

Introduction

After the complete human genome has been published by the Human Genome Project in 2001, there is a wide new range of scientific questions to be treated. Beside just knowing how the linear DNA-sequence looks like, we want to understand, how proteins are translated into complex structures and how they interact with each other. Therefore, it is crucial to understand what dynamic processes there are in cells and which influence they have got. One step to dive into this field is to be able to compute association rates between molecules. It is an indication for us if pairs of, e.g., proteins really interact with each other, in which medium and how often they build up a complex or get into new conformations in a dynamic environment. We also want to know how solvent, temperature or other environmental changes effect molecule-behavior.
With Brownian Dynamic simulations we are able to compute diffusional association rates *in silico* and simulate protein-protein encounter. In 1827 Robert Brown was the first one who observed pollen in water and described their motion. Later in 1926 A. Einstein described this dynamic motion of particles mathematically which have got a mass much higher than the one of its surrounding solvent [Ein05]. Collisions with solvent-molecules are described stochastically. This random movement is called diffusion [vS06]. Protein-protein interactions form important steps for example in membrane interaction, cell-cycle regulation, signal transduction or other regulatory mechanisms. Simulation of protein-protein interactions in physical environment is computationally hard. Therefore, we have to introduce into simplification. Solvent and auxiliary solute species like salt ions are represented as modifications in forces and by frictional and stochastic forces.
In this thesis we modify, evaluate and implement parallel algorithms which perform in parallel to border up Brownian Dynamics simulations. We formulate the sum-product problem and show that a speedup of ≈ 30 is possible for molecules with huge atom-sizes. Another boost with a factor of ≥ 80 is shown for force (resp. torque, energy) calculation if multiple BD simulations (≥ 100

BD simulations) run concurrently. An implementation of a parallel trilinear interpolation kernel is implemented and evaluated for preprocessing purposes and it is shown how to design algorithms which will be scaled up by future GPU devices.

This work is subdivided into four parts. At first, we will get into Brownian Dynamics and learn what they are and on which theory it is based on. Afterwards we will briefly get into an implementation of a BD-simulation called SDA (*Simulation Of Diffusional Encounter*). Later on, we will see how NVIDIA's GPU architecture works, get into its SDK called CUDA (*Compute Unified Device Architecture*), which libraries we are able to use and give some outlook in development. After the methods-section we will see which parallelization techniques are candidates for BD-simulations and how they work. In the last sections we analyze their performance and will discuss which hardware architecture is being required to safe computation time and make some comparison. We discuss our results and give some outlook in future development of BD-simulations and how efficiency can be optimized.

Chapter 2

Theory

In this chapter we will discuss how to calculate electrostatic forces and molecule energies based on the implicit continuum solvent model and dive into the Poisson-Boltzmann-Equitation. Later on, we will see how a BD-simulation is being implemented by a program called *SDA*.

2.1 Electrostatic Fields and Potentials

Electrostatic forces are very small but ubiquitous. They appear between ions (negative anions and positive cations), ion-dipole interactions (e.g. negative partial charge of water with positive cations) or dipole-dipole interactions (polar molecules like HCL). Also hydrogen bonds are preserved by electrostatic forces. We are able to describe electrostatic forces F like Newtons law of gravitation. Two charges Q_1 and Q_2 are described as

$$F = \frac{Q_1 Q_2}{K r^2} \tag{2.1}$$

This is the general Coulomb's law. Both charges are in a vacuum and at a distance of r. If $K = 1$ a charge unit (LE) is a electrostatic unit (esE). Two charges every 1 esE and at a distance of 1 cm are performing a force of 1 dyn to each other. Internationally a charge SI unit is Coulomb (C: $1C \approx 3 \cdot 10^9 esE$). So we write for $K = 4\pi\epsilon_0$ whereas ϵ_0 determines the permittivity or dielectric constant in vacuum, respectively. ϵ_0 is a measure for the transportability of electric forces in an electric field. In vacuum it has got a value of $8.854 \cdot 10^{-12} C^2 N^{-1} m^{-2}$ (or $C^2 J^{-1} m^{-1}$).

$$F = \frac{Q_1 Q_2}{4\pi\epsilon_0 r^2} \tag{2.2}$$

The electric field force E is defined as

$$E = \frac{Qr}{4\pi\epsilon_0 r^3} \tag{2.3}$$

E of any point determines the force which has to be performed to hold that point at its place. This equation shows that a field vector which has been aroused through a positive charge gets the same direction as a vector from the origin to the measured point. Therefore, we get the following norm for the field vector:

$$|E| = \frac{Q}{4\pi\epsilon_0 r^2} \tag{2.4}$$

To obtain all forces in a field, a summation of every resulting force through vector addition is needed. An electric field performs a force of 1 N to a charge of 1 C. Thus the unit of a vector field is N/C. But because $1VC = 1Nm$ a vector field has got the unit V/m.

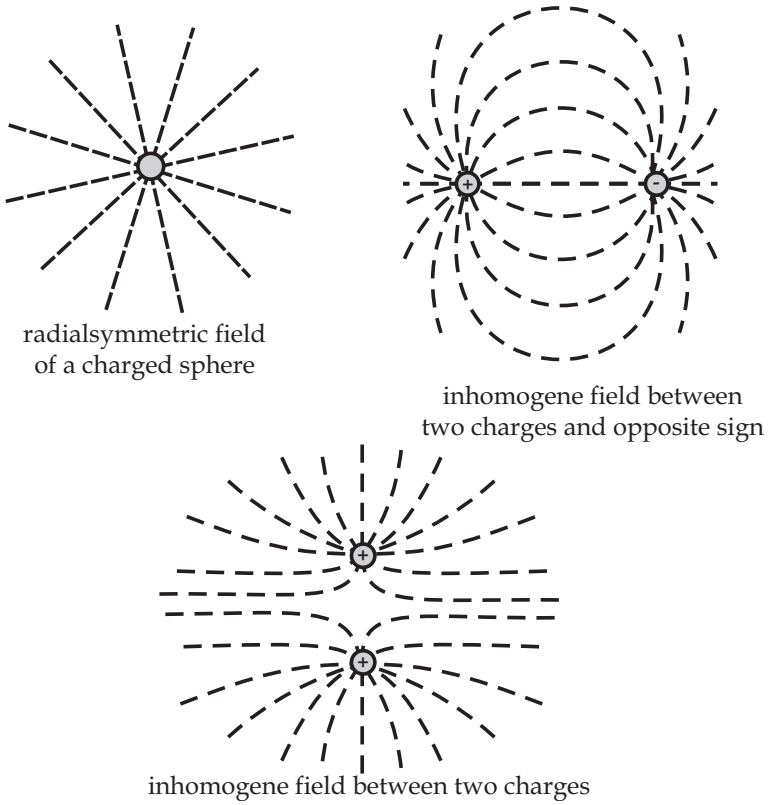


Figure 2.1: Electrostatic fields

In every point of a space we are able to display the electric field with discrete vectors if we move the test charges. Summarized, all vectors result in flux lines which are characteristic for vector fields (Figure 2.1). All field vectors describe tangents of the flux lines at every point of the space.

We can represent the electrostatic field as a gradient of a potential Φ.

Accounting all three components of the vector field, we get:

$$E_x = -\frac{\partial \Phi}{\partial x}, \quad E_y = -\frac{\partial \Phi}{\partial y}, \quad E_z = -\frac{\partial \Phi}{\partial z} \tag{2.5}$$

An example of a gradient is been shown in figure 2.2. A blue surface indicates a positive and a red surface a negative electrostatic potential.

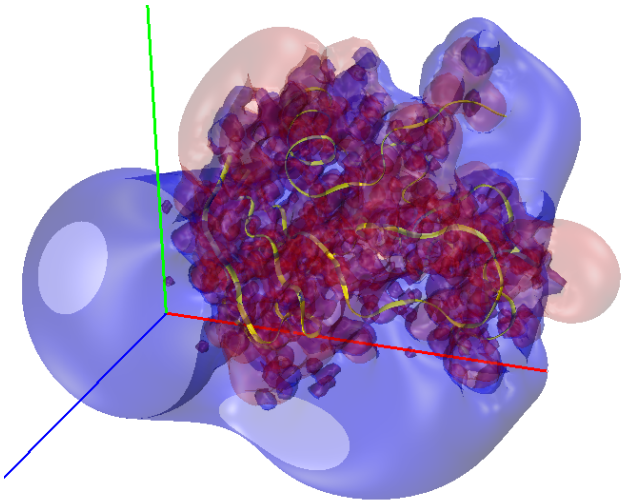


Figure 2.2: Gradient of barnase electrostatic potential

In a vectorial description of this relation is

$$E = -grad(\Phi) = -\nabla \Phi \tag{2.6}$$

A negative sign means that a positive charge would move from a higher potential to a lower one and we need to do work on getting a charge to opposite direction.

With this definition it is much easier to account calculations with a scalar potential function $\Phi(x, y, z)$. We are able to calculate a force in every point of a field.

2.2 Implicit Continuum Solvent Model

In electrostatics and especially for BD-simulations we use a continuum solvent model or implicit water model, respectively. An explicit modelling of water is computational hard and takes magnitudes of longer calculations compared to implicit water simulations, because a huge amount of water molecules have

to be simulated, too. Therefore, we will restrain to an implicit model which simulates water as a continuum with a high dielectric constant ϵ (Section 2.1). In this model, space is divided into an inner (*protein*) and outer molecule surface (*water*) which is called *solvent accessible surface* (Figure 2.3).

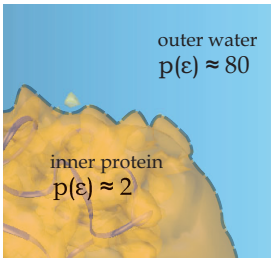


Figure 2.3: Continuum model with solvent accessible surface

Water has got a net charge of zero. This is why partial charges lie in the interior of proteins. These charges result to a charge density defined as $p(r)$ (Section 2.3).

2.3 Poisson-Boltzmann Equation

The Poisson-Boltzmann equation (PBE) describes electrostatic potential in ionic solvent. Because proteins are in ionic solvent we need to solve this equation to calculate electrostatics. This equation is important because it can be used to solve implicit solvent models which we described in section 2.2. It uses approximations on the effect of solvent on structures like proteins and other molecules in solution of different ionic strength. This equation consists of two parts. The first part describes the electrostatic field of ions and the second one describes the number of ions per unit volume in a particular region of space (Figure 2.4).

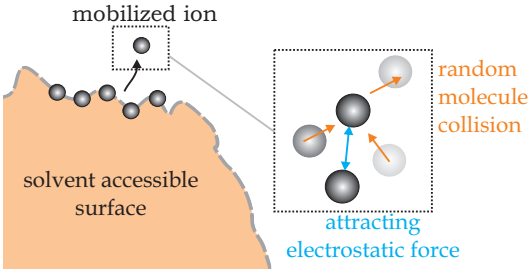


Figure 2.4: Ions approximated by PBE

The first part of the Poisson-Boltzmann equation is the Poisson-Equation

which is defined as

$$\Delta\Phi = \nabla(\epsilon(r)\nabla\phi(r)) = -\frac{p(r)}{\epsilon_0} \tag{2.7}$$

Interestingly only $p(r)$ and $\epsilon(r)$ are required to define the potential $\phi(r)$. $\epsilon(r)$ denotes the dielectricity function and $p(r)$. $\phi(r)$ and ϵ_0 are described in section 2.1.

In our setting permittivity is a discrete charge distribution of point charges and its value is defined by the function $p(r)$.
Dielectricity is a material specific value and is returned by the dielectric function $\epsilon(r) = \frac{\epsilon}{\epsilon_0}$. It is the ratio of the material specific permittivity ϵ to the permittivity of vacuum ϵ_0. With this value electrical properties of various materials are comparable.

A second term is needed to consider counterions, because positive and negative ions act like shields. Positive charged parts of a protein affect appealing to anions and negative ones to cations. This effect has been considered through the Debye-Hückel theory which describes anions as Boltzmann distributed and delivers an additional charge density:

$$-\kappa^2 \cdot \sinh\left(\frac{\epsilon(r)\phi(r)}{kT}\right) \tag{2.8}$$

κ is a modified Debye-Hüeckel parameter which describes ionic strength. For simplicity this equation is linearized with $\sinh(x) \approx x$:

$$-\kappa^2 \frac{\epsilon(r)\phi(r)}{kT} \tag{2.9}$$

This shielding charge density can be inserted into the Poisson equation. Therefore, we get the resulting Poisson-Boltzmann equation (PBE):

$$\Delta\Phi = \nabla(\epsilon(r)\nabla\phi(r)) - \frac{\kappa^2\epsilon(r)\phi(r)}{kT} = -\frac{p(r)}{\epsilon_0} \tag{2.10}$$

k is the Boltzmann constant and T is the absolute temperature which will be explained in more detail at the next section 2.4.
We are able to solve this equation with FDPB (finite difference PB, [BL92]) on a 3D grid by linearizing the equation and calculating $\Phi(r)$ at every grid point which have equal distance between 0.4 - 1.0 $\AA$ to each other. This is possible because we are able to define $\epsilon(r)$ (inner protein 2, outer water 78), κ and $p(r)$ for every crosspoint and interpolate between them. This method is used by APBS [NB01] (*Adaptive Poisson-Boltzmann Solver*) which calculates the PBE numerically.

2.4 Brownian Dynamics Simulations

Brownian dynamics algorithms simulate movement of microscopical particles in solvent. The movement is caused by thermal movement of liquid molecules. It was first detected by Jan Ingenhousz in 1785 who had interestingly also discovered photosynthesis. But like his other colleagues he thought the origin of the movement is of biological manner.

Finally, Robert Brown observed dithering movement of pollen in water and draw his first conclusion. Since then this effect is called Brownian motion. Later in 1926 A. Einstein described this dynamic motion of particles mathematically [Ein05] which have got a mass which is much higher than the one of its surrounding solvent. Collisions with solvent-molecules are described stochastically. This random movement is called diffusion [vS06].

With Brownian dynamics we are interested in protein-protein interaction and calculate association rates. These reactions have to be diffusion controlled. Diffusion control in a biochemical enzymatic reaction is the rate at which the enzyme can actually bind with its particular substrate. The upper bounds for the rate of enzymatic reactions is about $10^8 M^{-1}s^{-1}$ to $10^9 M^{-1}s^{-1}$. If the diffusion controlled rate is much higher than other chemical reaction rates, we are able to calculate the protein-protein interaction with BD-simulations.

We have to consider a lot of effects which influence protein-protein interactions in many biological processes but also want to compute these interactions computationally fast. For prediction of association rates between proteins we have to know that diffusion constrained rates does not reach the upper bound but are mostly $\geq 10^6 M^{-1}s^{-1}$. These association rates are sensitive to environment. Because they are dependent on the ionic strength of the solution, long-range electrostatic forces are important to account. Another important point for a diffusion controlled association rate is inverse dependence on solvent viscosity and linear dependence on the proteins' relative diffusion constant. Also dependency on temperature has to be taken into account.

Why are we using Brownian Dynamics?

In principle BD-simulations are similar to MD-simulations (*Molecular Dynamics*). MD-simulations simulate in the order of nanoseconds because they consider degrees of freedom and intramolecular forces but we want to be able to simulate a time range of milliseconds. Therefore, approximations are introduced for BD-simulations. Simulation of water, for example, is computationally hard and is approximated. It has got several important effects:

- Slowing down motion speed by viscosity
- Collision between water molecules and proteins

A BD-simulation uses an implicit continuum model to model the solvent which is explained in section 2.2. Thereby the calculation of thousands of water molecules are omitted.
Also another important approach is pursued. After test charges have been used in recent years, nowadays effective charges are computed. For this approach Razif R. Gabdoulline and Rebecca C. Wade use a clever rescaling of test charges modelling better physically related charges and showed that a better approximation is possible by using effective charges [GW96].

2.4.1 Principles

In classical BD-simulations two proteins are simulated. The first protein (PI) is fixed in the center of a coordinate system. A second protein (PII) is freely moving. For every step (BD-step) a calculation of PII's movement and rotation is made. This movement has to be considered relative to PI. After starting a simulation the algorithm has to parse charge and coordinate input-files of PI and PII. These files are processed and for example have modifications like some hydrogen atoms have been added or specific mutants have been modelled because we want to analyze their influence on association rate between PII and PI. After parsing an assignment of atomic charges and radii are made. Also surface atoms of PII are excluded and a construction of an exclusion grid for PI is been made. Next we have to compute the electrostatic potential of PI with for example UHBD [ea95] or APBS [NB01]. After this step we define a surface around PI which we call b-surface. A typical distance for the b-surface is 50 - 100 $\AA$. After placing PII on the b-surface (Figure 2.5) randomly, we have to define a so called escaping surface declared as q-surface.

The q-surface specifies a space around PII where PII should not be out of range. Normally, this surface has got a distance between 100 - 500 $\AA$. If PII is leaving the q-surface the simulation is aborting or restarting. After placing PII on the b-surface the most important step is performed. The algorithm starts to calculate the electrostatic force and torque of PII and multiplies it with a timestep Δt. This timestep depends on the distance of PII to PI. If PII is near PI, for example it can be inside the q-surface, Δt will be small and vice versa. After every step the algorithm has to check, if there is a so called encounter complex. An encounter complex is defined by the user and indicates a fit of both proteins or a pre-complex formation which defines a complex at a specific position where we are sure that this formation will perform a complex because for example it will rotate side chains and therefore allow some structures to get into contact with hydrophobic forces. For an encounter complex either we are able to define hydrogen bonds and their number which have to be established or we can define lists of atom-pairs which have to be in a predicted distances to each other. Another approach for defining an encounter complex is to calculate the RMS distance of selected atoms for both proteins. Because the number of

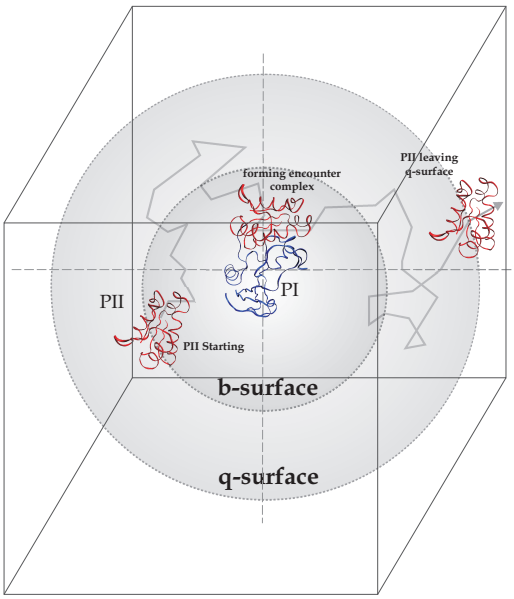


Figure 2.5: Setting up a BD-simulation to compute biomolecular diffusion-controlled rate constants

calculations grow extensively with larger proteins this approach is less used. A repetition of the movement for PII relative to PI is made until PII exits the q-surface. In this case the calculation of an association rate k is given by

$$k = k_D(b)\beta^{\infty}. \tag{2.11}$$

$k_D(b)$ is the steads-state rate constant for two particles which are in distance b. Distance b defines the distance where PII is placed onto the b-surface. β^{∞} is the probability for performing an encounter complex rather than diffusing away to infinity. Hence $k_D(b)$ is calculated analytically and β^{∞} is calculated by BD-simulations with different distances for b. Thousands of trajectories have to be calculated for this purpose. And because the force and torque calculations take a long time for huge proteins we have to minimize the time used for such calculations by trying to boost them with parallelization techniques on the GPU.

2.4.2 Translational and Rotational Displacements

Brownian motion in theory describes the dynamic behavior of particles. This random motion that is diffusion was described mathematically by Einstein and Smoluchowski [Ein05]. A particle which subjects to Brownian motion in time Δt has got an average displacement Δr which is defined as

$$\left\langle \Delta r^2 \right\rangle = 6D\Delta t \tag{2.12}$$

D is the translational diffusion constant of a spherical particle and is given by

$$D = \frac{k_b T}{6\pi\eta a} \tag{2.13}$$

η is the solvent viscosity and is a measure for an internal resistance to flow and is like fluid friction. For example water has got a viscosity η of $1.308 \cdot 10^{-3} Pa \cdot s$ at a temperature of 283 K and $\eta = 8.90 \cdot 10^{-4} Pa \cdot s$ at a temperature of 298 K. a defines the hydrodynamic radius of a particle. k_b is the Boltzmann constant which is defined as $1.3806504 \cdot 10^{-23} \frac{J}{K}$. This value is derived from following equation $k_b = \frac{R}{N_A}$ where $R = 8.314 \frac{J}{K \cdot mol}$ (gas constant) and $N_A = 6.022 \cdot 10^{23} mol^{-1}$ (avogadro constant). $6\pi\eta a$ defines the friction coefficient. We are able to use this equation because the radius of a diffusing particle is much higher than the radius of solvent molecules. If both radii are similar we have to use 4 instead of 6 as factor in the numerator.

For protein-protein interactions a method is needed which describes dynamics of diffusional motion. This can be solved by the Langevin equation. The Langevin equation is a stochastic differential equation in which two force terms have been added to approximate the effects of neglected degrees of freedom. A method to solve this equation was presented by Ermak and McCammon in 1978 [EDM85].
BD-steps are generated which reproduce current states of proteins at specific time intervals of Δt and result to trajectories. With the following equation we are able to calculate translational motion.

$$\Delta r = \frac{1}{k_b T} D F \Delta t + R_{\Delta r} \tag{2.14}$$

Where Δr is a translation vector. D and $\frac{1}{k_b T}$ also appear in equation 2.13. $\frac{1}{k_b T}$ describes the mean energy of an atom with temperature T and also has got a damping effect. F is the sum of all forces in the electrostatic field of all atoms of PII and has to be calculated every step Δt. $R_{\Delta r}$ is a random vector with $\langle R_{\Delta r} \rangle = 0$ and $\langle R_{\Delta r}^2 \rangle = 6D\Delta t$. This also models random motion in solvent. That is why we are not just accounting a directed force like the first summand describes.

Equally, rotational displacement with an angle Δw is defined:

$$\Delta w = \frac{1}{k_b T} D_r W \Delta t + R_{\Delta w} \tag{2.15}$$

D_r is the rotational diffusion constant, $R_{\Delta w}$ is a random rotation which fulfills $\langle R_{\Delta w} \rangle = 0$ and $\langle R_{\Delta w}^2 \rangle = 6D_r\Delta t$ and W is the torque which is the sum of all torques of every atom of PII. We get a vector with angles which is multiplied by Δt. Δt is variable and changes while BD-steps are calculated. It has got a smaller value if PII is near PI and a higher one if it is far away.

Smallest time steps are usually between 0.5 - 1.0 ps.
In the end our rigid body PII is translated by adding Δr and rotated by taking all angles of Δw and rotating PII with a rotation matrix for every direction in space.

Finally, we want to point out which limits of Brownian dynamics are given. Flexibility is not accounted yet and some interactions are not considered to make these simulations more accurate but also computational hard like allowing hydrophobic interactions between proteins although they were described by Donald L. Ermak and J. Andrew McCammon for BD-simulations in 1978 [EM78]. For association rate calculation of protein-protein interactions it has been shown with the interaction between Barstar and Barnase [Hel06] that performing BD-simulations the way we have described are delivering good approximations to real-world experiments but it is crucial to define correct reaction criteria.

2.5 Simulation Software

There are some resources for BD simulations which use similar approaches. One of them is Macrodox which is developed by Scott Northrup and colleagues. It is available at `http://iweb.tntech.edu/macrodox/macrodox.html`. Macrodox can be used to perform BD-simulations with a simple atomic-detail model. With this software it is also possible to approximate pK_a values. Another program is UHBD which has been mentioned before and is available at `http://adrik.bchs.uh.edu/uhbd.html`. Beside solving the PBE and other equations it can be used to simulate steady-state and non-steady-state association rates for protein-protein interactions. Therefore, they use a simpler model than an atomic-detailed one. Another approach is being made by E-Cell. It is a community who are building up a prototype and using a GFRD algorithm (Green's Function Reaction Dynamics) which was originally proposed by Zon and Wolde [ZtW05]. This prototype is available at `http://www.e-cell.org/ecell/software/bd-simulator-prototype`.
Another available software is SDA which was developed by Gabdoulline and Wade and which is able to solve simple or atomic-detail models and has got the ability to compute effective charges using the potential grid computed by UHBD and other enhancements. The authors provide a freely available Fortran source code. This is why we want to briefly describe how a BD-simulation with SDA is performed.

2.5.1 SDA Approach

SDA (*Simulation of Diffusional Association*) is a classical Brownian Dynamics simulator and is available at `http://projects.villa-bosch.de/mcmsoft/sda/`. The most important thing is that it is adaptive for a wide range of protein-protein interactions which are diffusion limited. We also have to define encounter states.
The program is divided into three parts. The first part consists of an initialization. It loads all necessary data like potential-grid, DX-grid files, from PI and PII, PQR-Files with atom-coordinates and charges and some configuration file to determine variables which have been mentioned in section 2.4. It makes an exclusion grid for PI and selects all surface atoms of PII to be able to detect collisions. After this, it reads the reaction criteria and builds up rules to be able to count an encounter complex after a BD-step. If there is a collision it will just discard the step and will retry to make a new BD-step.
The second part is a processing step. SDA writes out all constants and then shifts all coordinates to a computed center after it has read the input center.
The most important part of this algorithm is the third one. This part is repeated several times until PII leaves the q-surface (Figure 2.5). At first, PII will be positioned randomly on the b-surface and will get a random orienta-

tion. Now it will try to compute forces and torques. Hence it calls *gridforce* and sums up all forces and torques of selected atoms and *chargeforce* to calculate a force due to an effective charge. Now it has calculated a force which contains a directed and random component and a torque. The rigid body of PII will be moved and examined if there is an encounter complex. If there is an encounter complex the number of them will be incremented. If there is a collision *gridforce* will be called again and there will be no move until there is no collision anymore. Of course it can be hard to get out of this state so there is an approach to boost PII to get out of the "collision"-state. If there is an escape part three will be restarted. Now PII will be rotated. Again there is a check if PII and PI have got an encounter complex, a collison or if PII leaves the q-surface. Having an encounter complex it also calculates the energy of the system. If everything worked fine a successful step has been created and the algorithm returns to try computing a new step with new coordinates again. These steps are repeated until a maximal step size is reached or PII is out of the q-surface. In the end an association rate β^{∞} [GW98] and other statistics will be calculated and we are able to compare our results with experimental ones. All steps are summarized in detail in figure 2.6.

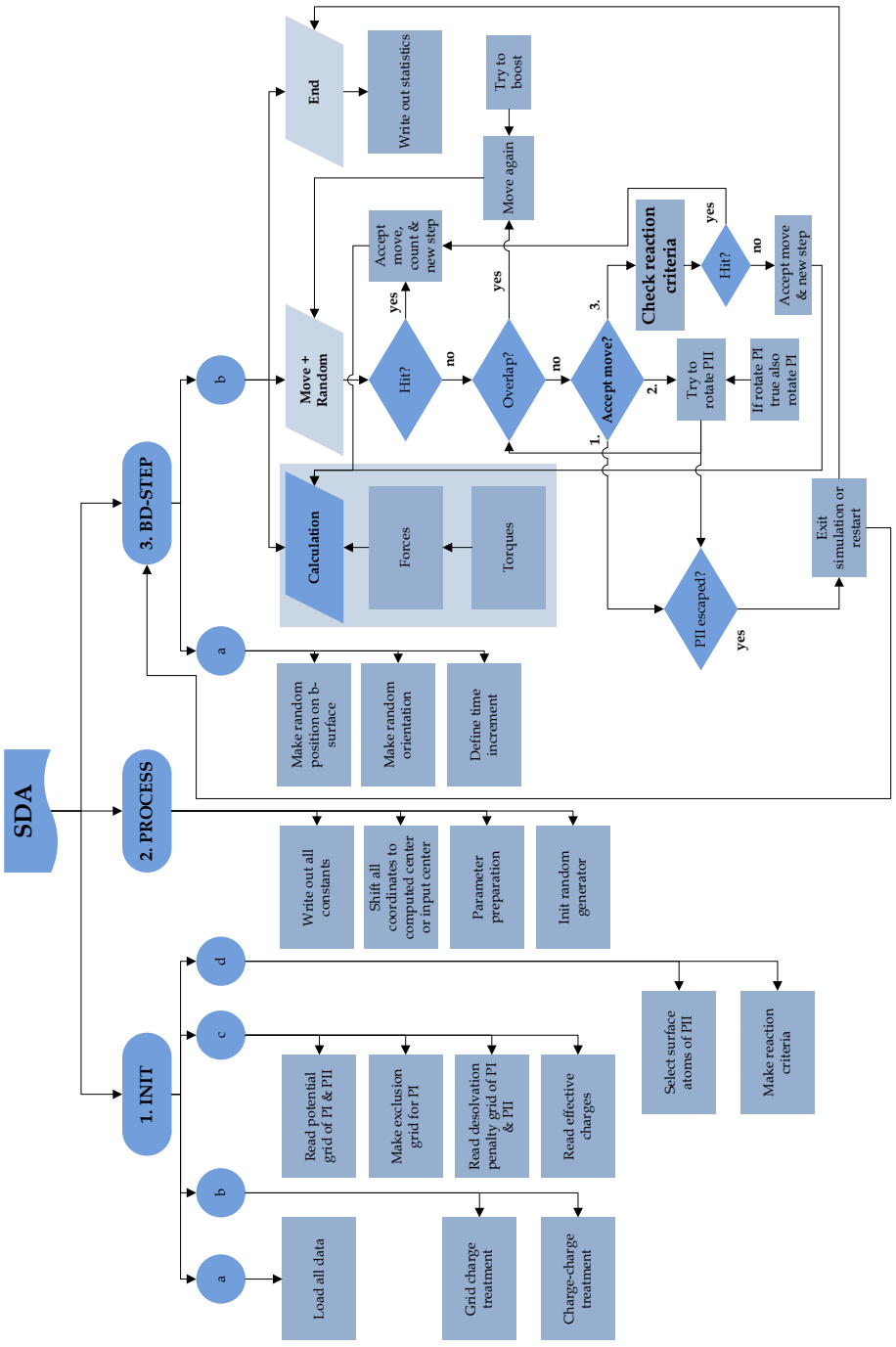


Figure 2.6: How SDA performs a BD-simulation

2.6 Compute Unified Device Architecture

2.6.1 Introduction

After NVIDIA was founded in 1993 it is one of the biggest manufacturer of graphic chips and chip sets for personal computer and game devices nowadays. Its headquarter is based in Santa Clara, California.
After producing their NV1, Riva and Vanta graphic chips they were most successful with their GeForce-series which still holds on. Now they produce the GeForce 8 series but are also preparing the 9^{th} generation. Because graphic cards with chips like 8800 GT/X are quiet achievable everyone can get a real powerful GPU (graphics processing unit) and compute with it. Every hardware has got a major and minor revision number which is called compute capability. The major revision number indicates the architecture (currently: 1) and the minor revision number shows which features are implemented (currently: x.0, x.1, and $\geq$ x.2). They are listed in table 2.1.
NVIDIA officially made their first CUDA SDK available to public in February

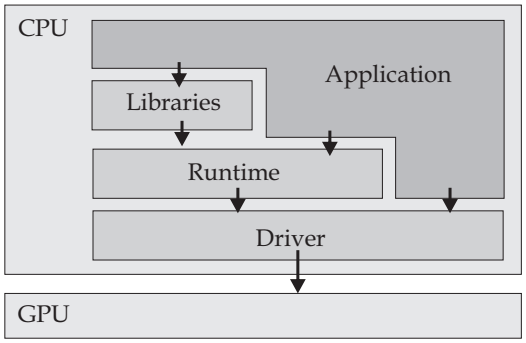


Figure 2.7: CUDA application access layer

2007. CUDA (*Compute Unified Device Architecture*) is a software development kit which enables access to driver, runtime and library components (Figure 2.7). CUDA is also directly compatible with OpenGL and DirectX but is not the only GPU architecture on market. Comparison between architectures is made by their performance on calculation with floats in GFLOP/s. As we can see in figure 2.8 NVIDIA's G80x architecture is providing a fast calculation capability compared to other approaches like AMD's "close to metal"-technology and an architecture called R600. Another approach is made by Intel. They are developing a multi-core graphics processor which is called "Larabee" and will be available in late 2009. Because of this competition we can expect in the near future to get chips which have higher computational power. With a theoretical maximum of nearly 1 Teraflop in the next year (Figure 2.8) there is room for improving a lot of simulation software and this is the goal in which

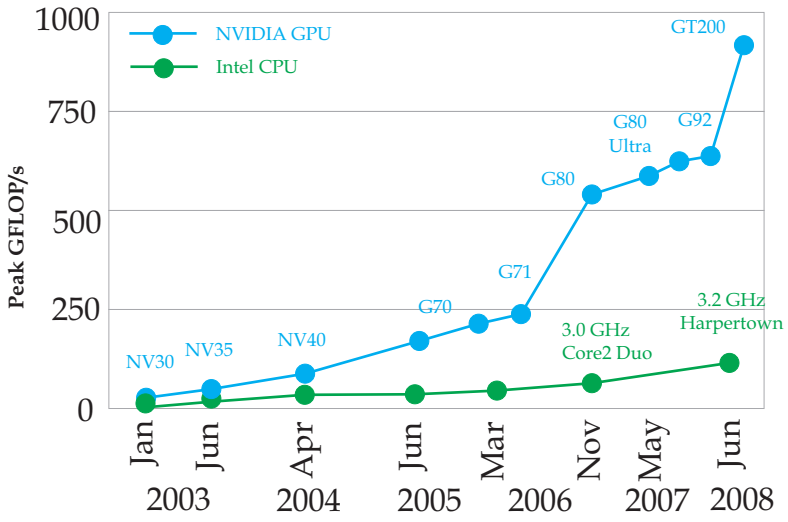


Figure 2.8: GFLOP/s performance CPU vs. GPU

this thesis is based on. We want to be able to perform a BD-simulation very effective by making use of parallelization techniques.

2.6.2 Architecture

Programming with CUDA is efficient if the underlying hardware is efficiently used. Additionally, algorithms have to be changed to fulfil performance requirements and to have a high occupancy. The GPU is a compute *device* which executes a high number of threads in parallel where every thread performs same calculations on different data. The CPU is called *host* and is facilitated by the device if there are massive calculations with low data load and high data reuse.
To use the device an algorithm is divided into many portions which will be calculated by threads that are organized in grids. In the end all results from every thread will be sticked together and results can be used by the host or are further processed by the device. A function which is executed on the device is called a kernel. A kernel is launched in a way that a specific number of threads are launched in parallel and with high processing balance to get the most performance out of the GPU. There are some locations for storing information on the host called *host memory* and according to this memory on the GPU is called *device memory*. It is important to consider that we have got different memory locations and therefore algorithm design choices have to be made for performance issues which will be explained in more detail in section 2.6.4.

Programming Model

A general aim is to highly parallelize program execution and solve several real world problems in a fast way. We always launch programs on the CPU for solving a wide range of problems. Generally, every code is executed sequentially on the CPU (Figure 2.9). The instruction units on the left are calculated sequentially and through the C-code it is possible to call a kernel function. A

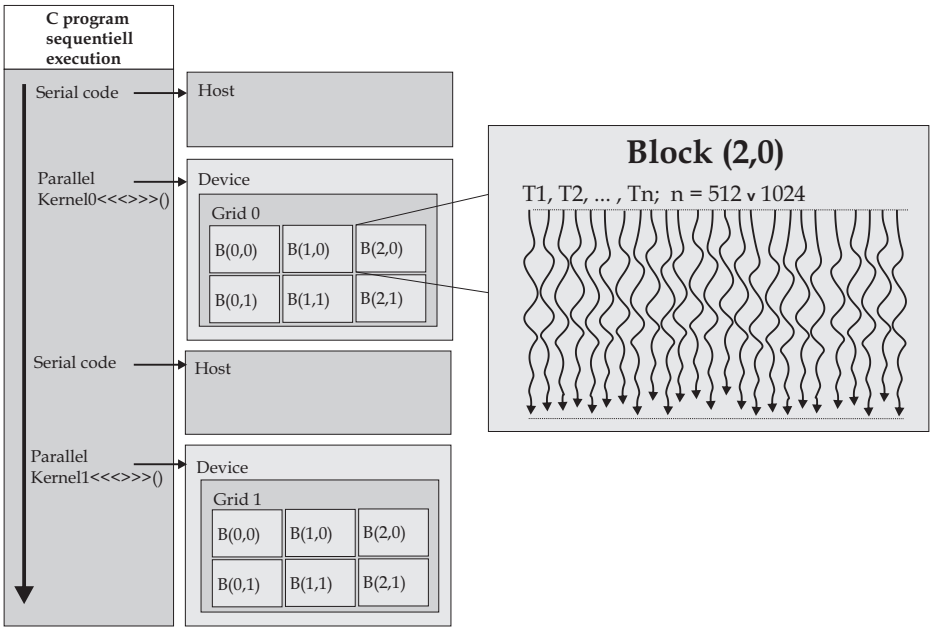


Figure 2.9: Execution model

kernel function is launched on the GPU. As drafted a kernel-launch executes grids on the device. Every grid is divided into blocks and every block in turn launches several threads and waits until they are processed. Therefore, the basic steps on implementing an algorithm on the GPU are the following ones. On the CPU we have to:

- Allocate data in GPU memory,
- Copy data from CPU to GPU,
- Invoke kernel with N threads and pass pointers to the memory arrays for association and writing issues at the first step,
- Wait until completion and copy data from GPU to CPU[1].

[1]This step can be enhanced through asynchronous data transfers while calculation on the GPU is not disrupted.

On the GPU side following approaches are made:

- Get ID for thread and determine on which data to work on.
- Start calculations on each thread and write output back to global memory.

Launching a kernel is computational cheap but to get most performance out of the GPU, grids and threads have to be loaded in a manner that all of our threads are in high occupation[2]. A low occupation means that we have to try making threads not to diverge into different code sequence separations for example by having different paths through if-decisions. We also have to hold threads busy by long calculation on the same data to get a high occupancy for all of our threads.

Hardware Model

Every GPU of the 8^{th} GeForce generation and further consists of N Multiprocessors ($N = 16$ for GTX8800, $N = 16$ up to $4x16$ for Tesla; Figure 2.10). This

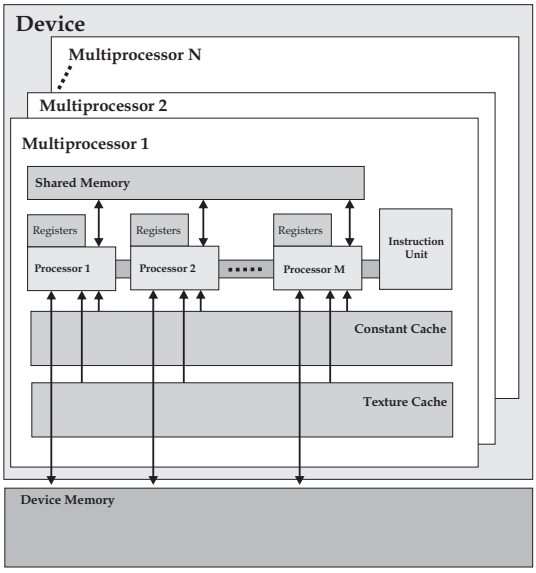


Figure 2.10: Hardware model

multiprocessors are controlled by NVIDIA's driver and are optimized for high computation where the performance is measured in GFLOPS. Furthermore, every multiprocessor consists of M processors. Every processor has got his own registers (v1.1: 8192, $\geq$ v1.2: 16384; versioning was described in section

[2]Occupation is a measure for activity of each thread in percentages.

2.6.1), shared memory (M x 16 kB) and local memory (Figure 2.11). Threads in blocks are able to access global memory which is not cached. Their size varies between 768 MB - 1.5 GB (up to 4GB in Tesla C1060).

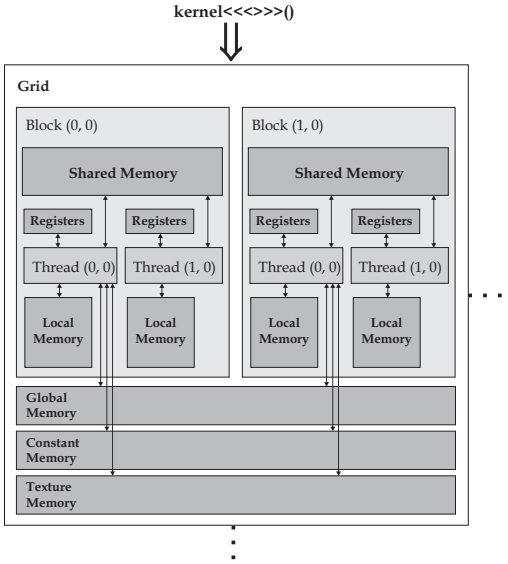


Figure 2.11: Programming and memory model

Each processor also has access to a constant memory and texture memory. Both have got a size of 64 kB with 8 kB cache but are read-only. They are fast if data is cached ($\approx$ 4 cycles) or if not cached as slow as global memory with a latency between 400 - 600 clock cycles, because data has to be fetched externally. Maximal 8 processors (= ALU, arithmetic logic unit) are activated and running concurrently if we have got a high occupancy which means that all running threads are in high computing activity. In the software model this processors are declared as blocks. One block consists of several grids which can be indexed one-, two- or three dimensional with a range of 65535 for each dimension. Threads in a block are indexed as 512 x 512 x 64, respectively. For example the index of a linear array of threads would be declared as: $thread(x, y, z) = (x + yD_x + zD_xD_y)$ where D_x, D_y denotes the block IDs if we define a 2D array. Warps are a collection of 32 threads and are launched together in every block. Actually they are build up by half warps and have got an upper and lower component (first 16 threads and later 16 threads in one warp).

A GTX 8800 has got 16 multiprocessors and 8 blocks running concurrently. That is why a GTX8800 has got 128 cores (16 multiprocessors x 8 blocks) which are able to operate concurrently on different data.
As mentioned before, we need a high occupancy for all of our threads because we want to touch the peak of $\approx$ 570 GFLOPS (GTX8800). This can only be

achieved if our algorithm is optimized for this design. In general, this is very difficult and a theoretical performance which is not reached fully in practice yet. The number of warps in one multiprocessor is maximal 24 which are running simultaneously. This means that the number of threads which are running concurrently are 24 warps x 32 threads/warps = 768 threads.

We are able to launch a huge number of threads. But not all threads are able to communicate with each other. Only threads of one block can communicate with each other through divided shared memory (Figure 2.11). Threads from blocks have to communicate with other blocks through global memory (constant and texture memory are read-only). This communication has to be kept as small as possible, because an access (read-write operations) takes up to 400 - 600 cycles per clock compared to 4 cycles in shared memory. That means that global memory operations are 125x slower than operations on shared memory. Too frequent access on global memory is leading to a worse performance. One also has to keep in mind that a concurrent write to a same area is not prohibited but the result is undefined. In this case NVIDIA guarantees at least one success operation. This means if threads want to gain access to the same memory at the same time this would cause a false result. This can be solved by functions which are atomic operations. An access is blocked until the memory is deallocated again. Such operations are serialized by a thread-scheduler and therefore a programmer has no direct influence to such behavior.

Summarizing for architecture with version 1.x we have to design algorithms in a manner that high parallelization with high occupancy and low divergence for each thread is guaranteed. This is possible if we load few data into threads and have a lot of calculations before writing our results back to global memory.

Available Hardware

We have mentioned precise memory sizes, operation times and quantities for threads, multiprocessors, etc. They are changing from time to time and precise values are given in the CUDA Programming Guide 2.0 [Cor08]. The higher the compute capability is the more features are implemented to be used and efficiency improvements are being expected.
An excerpt is given in table 2.1. Because of architectural advances it is possible to gain a higher performance out of our final code without editing any source code. An upscaling by using better hardware is possible because even if new features are not used, a trend will be to launch more blocks, grids and therefore threads concurrently for example by using more multiprocessors. This is possible because generally data is separated the way that it does not matter in which order blocks are launched. If some blocks take longer time to calculate there will be enough resources to start further blocks or terminate warps and half warps and launch them again on other reassures.

	M **multiprocessors**	**Compute capability**
GeForce GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce 9800 GTX	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0

Table 2.1: Technical specifications

2.6.3 Interplay with C

After installing CUDA, its compiler *nvcc* is used to compile CUBIN files (ending with cu). C and C++ files are compiled with a favored c-compiler like *gnu c* or other ones and all resulting object-files are linked together. We get an executable file which is launched on the CPU with some binary code for the execution of kernels in the GPU.

CUDA is made for coding with C and therefore a lot of C functions from the standard library are provided by the common runtime component. It consists of a host component which provides functions to control and access compute devices from host. Another part is a device component which provides device-specific functions which has got a common component and serves specific datatypes (Section 2.6.3) and a subset of the C standard library. Both, datatypes and the subset of the C standard library, are supported from host and device.

Datatypes

CUDA allows us to use vector types which are basic integer and float types. They are supported from one to four dimensions: char1, .., char4, int1, .., int4, float1, .., float4 and many more are implemented to be used [Cor08]. Another datatype is given by CUDA arrays which are defined as one, two or three-dimensional arrays to build up textures. They have got several functions like interpolation (3D interpolation is supported by CUDA 2.0), normalization, filtering and other texture-specific functions for graphic development purposes.

NVIDIA developers also promise fast access to these arrays because CUDA array data is loaded into vector memory which has 64 kB with 8kB cache.

2.6.4 Performance Issues

Accessing Memory

The efficiency of accessing memory, especially working with global memory, depends on access patterns. Access patterns describe reading and writing operations on memory which have to fulfill several requirements to have few access to global memory. Because of high latency of device memory compared to on-chip memory like shared memory, device memory access should be reduced. Hence we should do following steps:

- Load data from device memory to shared memory,
- Synchronize all threads after reading from device memory so that every thread is able to access shared memory safely,
- Perform calculation on data in shared memory,
- Synchronize again to be sure that all results have been written in shared memory,
- Write results back to device memory.

Coalesced Access Patterns

To create desirable access patterns we have to try accessing memory coalesced. Coalesced memory access means fetching variables in memory concurrently. This reduces the number of device access and increases memory bandwidth.
It is important to know that 32-bit, 64-bit, or 128-bit words from global memory are loaded into registers by a single instruction. Every word has to be a type of a size which is equal to 4, 8 or 16 bytes. Also their position has to be a multiple of their typesize. For example: A float is 4 bytes and fulfills requirements. A float4 datatype (Section 2.6.3) also complies requirements because a float4 has got 4 floats and 4 x 4 bytes = 16 bytes.
Another important approach to use bandwidth most efficiently is if a halfwarp (16 threads) can be coalesced into a single memory transaction. A single memory transaction has got a size of 32 bytes (for $\geq$ v1.2), 64 bytes, or 128 bytes. This means, if we have got 4 byte words which are fetched by a halfwarp, this results into a 64-byte memory transaction (4 bytes x 16 threads = 64 bytes; v1.1). 8 byte words result to one 128-byte memory transaction and 16 byte words into two 128-byte transactions. All 16 words of our half warp with 16 threads have to be at the same segment of memory transaction size. An access also has to be sequenced. That means that every i^{th} thread must access the i^{th} word. If this requirements are not fulfilled another or many more additional memory accesses have to be issued.

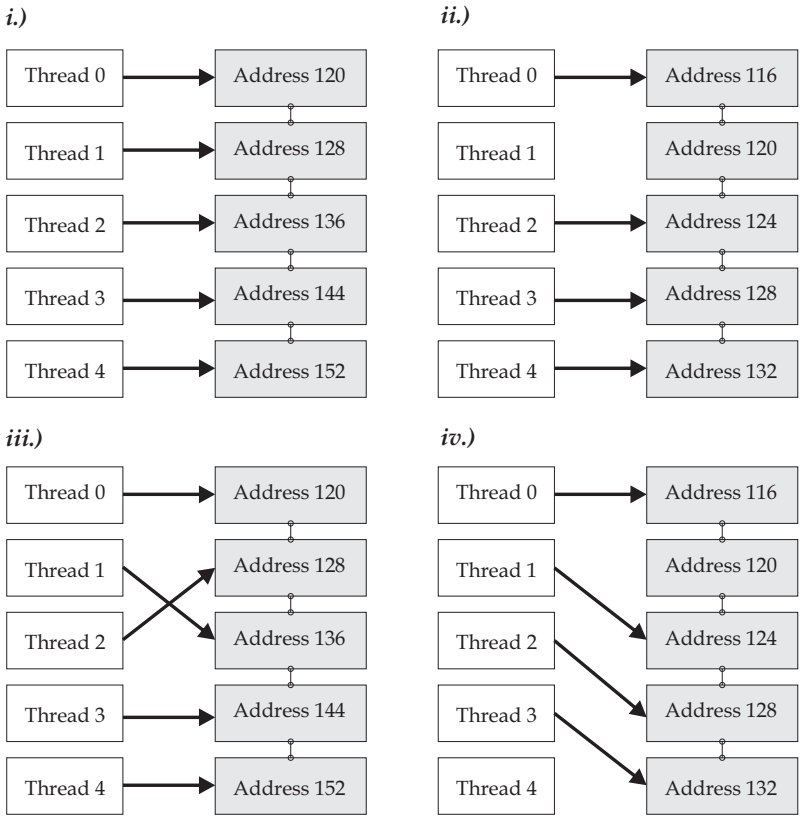


Figure 2.12: Access patterns: *i.* type = float2, coalesced access pattern; *ii.* type = float, coalesced; *iii.* type = float2, non-coalesced; *iv.* type = float, non-coalesced.

Also memory bandwidth would be significantly reduced. Compute capability cards with $\geq$ v1.2 are able to transact areas of memory where not every i^{th} thread must access the i^{th} word. This is very useful for some algorithms because otherwise they would need i separate transactions in worst case. Figure 2.12 shows typical coalesced and non-coalesced situations. In this figure *i.* and *ii.* are in coalesced access because every i^{th} thread accesses the i^{th} word. *ii.* is also possible although thread 1 behaves divergent. *iii.* is non-coalesced because not every i^{th} thread accesses the i^{th} word. Thread 1 has to access address 128 and thread 2 should point to address 136. At *iv.* address pointing is shifted and access is not sequential.

Avoiding Bank Conflicts

We also have to consider fetching variables from threads. Every thread executes same commands but loads different data to work on. Hence all threads have to load variables into shared memory concurrently and while considering memory access patterns described before we also have to pay attention to

memory bank conflicts.

Every block accesses shared memory. Currently ($\leq$ v.1.3) shared memory

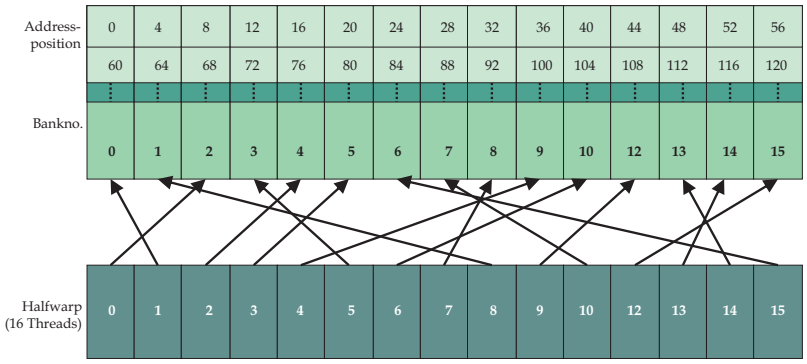


Figure 2.13: Bank conflicts. This example shows no bank conflict because access to banks by threads are bijective.

has got a size of 16 kB for each multiprocessor. This is why it does not matter how many threads are executed concurrently in one block which are organized in half warps for access pattern manner because all max. 512 threads for one block have to share this 16 kB. Therefore, we are able to boost thread scheduling and memory access to same areas of shared memory if we are able to separate access to this locations for every thread. This locations are called *banks* and are organized in 16 blocks for a 16 thread-access by a half warp. Furthermore, this blocks are divided into 1 kB areas which should be accessed by only one thread to have no bank conflict. This results in an access by threads without having to be serialized if threads try to concurrently access their bank. An example is given in Figure 2.13 where an array of 4 byte-words is saved. Thread 0 accesses bank 2 which has got positions of 8 + 60x and thread 1 accesses bank 0 which is attributed to positions 0 + 60x and so on where $x \in \mathbb{N}, x \geq 0$.

Loop Unrolling

Another performance optimization strategy is loop unrolling. Loop unrolling performs an operation such as described in Figure 2.14. If we consider loop

before	**after**
for (int i = 0; i <= 3; i++) A(i) = B(i) + C(i)	A(0) = B(0) + C(0) A(1) = B(1) + C(1) A(2) = B(2) + C(2) A(3) = B(3) + C(3)

Figure 2.14: Loop unrolling. This example shows what loop unrolling actually does.

unrolling and tell the compiler how to unroll a loop the compiler will be able to reduce overhead for initializing a loop because it will not have to consider structures like managing the loop variable and make queries if loop condition is fulfilled to repeat loop statements. Registers are used efficiently and therefore resources are saved or used more efficiently. For example a GTX 8800 has got 8192 Registers. This is why a loop could be unrolled in such a manner that registers are optimally used because accessing registers is very cheap and is as fast as fetching a value from shared memory. If a loop cannot be unrolled such that it fits into registers it has to be partly unrolled as shown in Figure 2.15. In this case the compiler tries to partly unroll the loop but for a small N loop

before

```
for (int i = 0; i < N; i++)
   A(i) = B(i) + C(i)
```

after

```
balance = N % 4

for (int i = 0; i < Rest; i++)
   A(i) = B(i) + C(i)

for (int i = Rest; i < N; i = i + 4) {
   A(i)   = B(i)   + C(i)
   A(i+1) = B(i+1) + C(i+1)
   A(i+2) = B(i+2) + C(i+2)
   A(i+3) = B(i+3) + C(i+3)
}
```

Figure 2.15: Partly loop unrolling if registers are too small to consider all control-structures.

unrolling is worse. That is why we are able to tell the compiler not to unroll loops for small N. With loop unrolling in mind it has been shown in several algorithms that accounting loop unrolling we get a better performance. Lars Nyland, Mark Harris and Jan Prins for example showed that performing an N-body simulation with 16384 bodies an increase from 184 GFLOPS to 204 GFLOPS are possible just by unrolling a loop by a factor of 4 [LNP07].

Performance Consideration for BD-Algorithm

Because we want to boost BD-algorithm it is evident to account all discussed points. That is we have to take care of half warps which fulfill access pattern issues to get a high memory bandwidth and avoid bank conflicts because we do not want to have serialized shared memory access of threads which reduces occupancy (Section 2.6.4). In the BD-algorithm our bottleneck is given by accessing memory. Therefore, we should use arrays like CUDA arrays which are saved in texture cache and provide cache to boost access because every matching request is up to *125x* faster than one access to global memory in worst case. We should then try to calculate as much as possible with our

threads (forces, torques and energy) to get a high GFLOP rate. A next step is a summing of all thread-calculations which will represent one atom each. One approach is to have a register value which will be summed up by results of threads of one block and then this result will be summed up with a global memory value for every result of every block. This is not really possible because although we get true results from every thread for a specific atom the sum of values will be wrong because a lot of threads will always try to write to the sum concurrently. In this case we are not able to handle access and it is undefined what results will occur. In this case NVIDIA promises that at least one operation will always be performed if there are concurrent write requests. This is a general problem handling a huge amount of threads which produce subresults for a general common result. In this case we need atomic operations. These operations solve this problem by serializing access to values on shared memory or global memory. Therefore, every operation is guaranteed but we have to consider performance loss. On the other side atomic operations are not available for all compute capabilities. Atomic operations on global memory are just supported for $\geq$ v.1.1 and atomic operations on shared memory are supported for devices of compute capability $\geq$ v1.2. Hence another approach is to fill arrays with results and sum them up faster than $O(N)$ in $O(log(N))$ time. We will evaluate various approaches in chapter 3.

Occupancy Calculations

In this last subsection we will briefly introduce occupancy calculations. With this calculations it is possible to get a fast way to decide how many threads and blocks (*dimensions*) should be used to call a kernel. For this purpose NVIDIA provides a sheet which calculates necessary dimensions [Inca].
If we define tpb_v as the number of threads per block, rpt_v enumerating registers per thread and $smpb_v$ indicating the size of shared memory allocation per block in bytes all formula which hold are given in Appendix 7.2. For example, if we define following variables:

$$\begin{aligned} tpb_v &= 256 \\ rpt_v &= 10 \\ smpb_v &= 4096 \end{aligned}$$

We will get following results:

In this example we see that we would launch 8 warps per block which denote $8 \cdot 32$ threads $= 256$ threads. It is the same value we have defined for tpb_v. We also defined $rpt_v = 10$. Because we have got 10 registers per thread and 16 threads are one halfwarp which denotes that $2 \cdot 16$ threads are one warp and we have got 8 warps running concurrently we get a total

component	**abbreviation**	**result**
Warps per block	wpb_*	8
Registers per block	rpb_*	2560
Shared memory per block	$smpb_*$	4096
Blocks per multiprocessor limited by warps	wpm_*	3
Blocks per multiprocessor limited by registers	rpm_*	3
Blocks per multiprocessor limited by shared memory	$smpm_*$	4
Blocks per multiprocessor	$atbm_*$	3
Active warps per multiprocessor	$awpm_*$	24
Active threads per multiprocessor	$atpm_*$	768
Maximum simultaneous blocks per GPU	-	48
Occupancy of each multiprocessor	-	100%

Table 2.2: Example of occupancy calculation, see Appendix 7.1

amount of registers per thread block of $10 \cdot 2 \cdot 16 \cdot 8 = 2560$. $smpb_v$ is defined as 4096 which means that every block is able to access 4096 bytes of data. Because every multiprocessor has got 16 kB shared memory and we have calculated that 3 to 4 blocks are running concurrently (we will choose 4 for clean values) $16 \cdot 1024/4 = 4096$ bytes for each block. If we have got 8 warps per block and maximal 24 warps are able to run concurrently we get $24/8 = 3$ blocks running concurrently if limited by warp size. We also know that every multiprocessor has got 8192 registers. This means, if one thread has 10 registers associated and we are running 256 threads this means we will get $floor(8192/(256 \cdot 10)) = floor(3, 2) = 3$ running blocks concurrently if limited by registers. Last but not least, if the number of concurrent running blocks is limited by the size of shared memory we will get 4 concurrent running blocks, because every block has got 4096 bytes of shared memory. This means, if one multiprocessor has got 16 kB, we have to calculate $16 \cdot 1024/4096 = 4$ concurrent running blocks. We want to give a lower bound, therefore we accept the minimum of concurrent running blocks which we have calculated so far and which is 3. The number of active warps per multiprocessor has got an upper bound of 24. But in this case the same value is calculated by having 3 blocks and 8 warps running in every block which is $3 \cdot 8 = 24$. The number of active threads per multiprocessor is 768 which is also an upper bound given by architectural purposes. This value is calculated by having 3 blocks running and 256 threads executed in every block which results to $3 \cdot 256 = 768$.The maximum simultaneous blocks per GPU is 48 because a G80 has got 16 multiprocessors and if every multiprocessor has got 3 running blocks this determines to $16 \cdot 3 = 48$.
Because of the occupancy of each multiprocessor is given by

$\frac{\text{active warps per multiprocessor}}{\text{maximal allowed warps per multiprocessor}}$ we get a value of $\frac{24}{24} = 1$ which denotes 100% occupancy for every multiprocessor in this example.

2.6.5 Libraries

Libraries support work with CUDA therefore we want to know which methods are provided and are useful for our purpose. This is why we want to introduce the most important libraries for CUDA.

CUBLAS

The CUBLAS (Basic Linear Subprogram) library is freely available for CUDA [Inca] and provides some subprograms which are optimized for NVIDIA GPUs. It allows access to computational resources and no direct interactions with the CUDA driver is necessary. CUBLAS provides creating matrix and vector objects, operating on them by functions like filling objects with data calling sets of CUBLAS functions and some helper functions are also provided for writing and retrieving data from these objects.
Most functions calculate with single-precision for fast runtime and are divided in BLAS2 and BLAS3 functions. BLAS2 functions provide matrix-vector operations which we will use in methods (Section 3) and BLAS3 provides matrix-matrix operations.

CUBLAS Fortran Bindings

Because many applications are written in Fortran and they would benefit from using CUBLAS, Fortran bindings have been implemented to make CUBLAS as easy callable as from C or C++ applications. Furthermore, it provides a lot of wrapper functions for maximum flexibility in addressing a lot of differences even in Fortran code. It could be possible to boost SDA (Section 2.5.1) by performing some calculations on the GPU. For further references see Appendix A of a technical guide "CUDA CUBLAS Library" which is available at NVIDIA [Inca].

CUDPP

CUDPP which stands for CUDA Data Parallel Primitives Library is a library which includes some important algorithms like various scan algorithms (e.g. parallel-prefix-sum). Also sparse matrix-vector multiplication and parallel reduction are implemented with a Plan-interface which is similar to other libraries. We will use the matrix-vector multiplication and parallel reduction

primitives to gain some performance measurement in the next chapter (Section 3). CUDPP is also freely available [Lib] and is expanded by an online-community.

2.6.6 Parallelizability on CUDA

In the prior sections we have shown how NVIDIA has build up a GPU architecture, how access patterns are performed, discussed how to avoid bank conflicts and pointed out occupancy calculations. If we take care of all discussed points in algorithmic design we will get very fast algorithms which are boost up compared to a CPU implementation in a high magnitude of orders. NVIDIA shows such approaches and we are able to discover boosts in calculation from 13x speedup (Biomedical Image Analysis) until 260x (Accelerating Statistical Static Timing Analysis). All of this algorithms have got some similarities:

- Fewer read-write access to global memory and effective access pattern,
- Working with cached memory,
- Designing an algorithm by subdividing it into small parts. This parts have to be calculated very efficiently in a massive parallel way,
- A lot of calculation is made on few data,
- The size of threads which are needed has to be very high (sometimes $\geq$ 65000) to get a very high memory bandwidth and GFLOP rate.

As it becomes clear not every scientific problem is amenable considering these points. But mostly, if an algorithm is able to be designed by fulfilling some of these requirements a high boost is likely to be expected. For some example application see [Incb].

2.6.7 Future Development

NVIDIA has shown that it is possible building up an architecture which is scalable in the future and speeds up algorithms which are designed for this architecture. The GPU market is highly embattled and competing firms like AMD and Intel also try to settle into the HPC market (*high performance computing*) where Intel dominated the market for further years. Because of this competition it is expected to get better processors in the coming months and years. We also have to take care about other architectural approaches like the CELL-architecture which prognosticate a peak of 1 TFLOP in early 2010 [paL]. This architecture which was originally developed by companies like IBM, Sony and Toshiba are also providing parallel execution components

called SPEs (Synergistic Processor Element), have got an own cache and are operating isolated until calculations are finished. Because of an interesting DMA transfer and a local memory size of 256 kB such an architecture could also be interesting if a high read-write access for an algorithm like the BD-algorithm is expected. But because of a large community and nearly daily progress we expect that NVIDIA has produced an architecture which has got a chance to affirm in the near years for the HPC market.

Chapter 3

Methods

In this chapter we will introduce some parallelization strategies which are used in the next chapter to improve the BD-algorithm (Section 4). At first, we show how to calculate important parameters like forces, energies and torques and how to approximate the potential $\Phi(r)$ of every atom with position r by trilinear interpolation. After this, a Parallel prefix sum is performed and we show which part of this algorithm is used. Later on, Parallel sum reduction is evaluated and matrix-vector multiplications are calculated. For that purpose, different libraries (Section 2.6.5) are used.

3.1 Requirements for a BD-Step

At first, we briefly introduce how to perform calculations for moving a protein (PII) relative to protein (PI) in every BD-step (Section 2.4.1).

3.1.1 Forces, Energies and Torques

A regular way for defining a force F is given by Newtons law: $F = m \cdot a$. F is a force, m is the mass of an object and a is the velocity of an object. Because we have got atoms with position r the equation is equivalent to

$$F(r) = m \cdot a(r) \tag{3.1}$$

$a(r)$ is described as a gradient of an electrostatic potential field $-\nabla\Phi(r)$ which is described in equation 2.6. It is possible to solve $\Phi(r)$ by APBS on an equidistant 3D grid numerically. Therefore, we can get an approximation $\Phi(r)$ by trilinear interpolation (Section 3.1.2) for every atom-position of our protein in the 3D grid. In an electrostatic field charges q_i are used instead of m_i. In our case, q_i defines the charge of the i^{th} atom. As charges we use effective charges

because they are good approximations for real charges of an atom [GW96]. Hence the force F for an atom at position r is

$$F_i(r) = m_i \cdot a(r) \equiv q_i \cdot -\nabla\Phi(r) \tag{3.2}$$

Every component of $\nabla\Phi(r)$ has to be multiplied with the appropriate effective charge q_i (Section 2.4) before summation. The interpolation and summation of all forces for every atom is a bottleneck of the BD-algorithm because the amount of sum-products increases by increasing atoms.

To get the overall force at every BD-step, we have to calculate $\nabla\Phi(r)$ for all atom-coordinates and sum them up to get a force-vector which moves the rigid protein PII for a specific timestep Δt.

$$F_{all} = \sum_i m_i \cdot a_i \equiv \sum_i q_i \cdot -\nabla\Phi_i \tag{3.3}$$

To rotate a rigid protein, we also have to calculate torques w_i for every i^{th} atom and build an overall sum. A torque is a vector which defines a tendency of a force to rotate an object about some axis. A protein will be translated if it is not hold at a fixed place. But if it is fixed and a force is acting on it, the protein will rotate around the fixed axis (normally a pre-calculated center).
A torque w_i is defined as:

$$w_i = r_i \times F_i \tag{3.4}$$

The result is a cross product of r_i and F_i. r_i is the position of the atom as a vector and F_i is the force affecting the i^{th} atom (Equation 3.2). Hence the overall torque for the protein PII is given by

$$w_{all} = \sum_i (r_i \times F_i) \tag{3.5}$$

For a final translation and rotation of PII we have to calculate Δr and Δw which is described in section 2.4.2.

We get following equation for the energy E of an atom at point r:

$$E(r) = \frac{1}{2} \cdot q_i \cdot \Phi(r) \tag{3.6}$$

Therefore, the overall electrostatic energy of a protein is given by

$$E_{el} = \frac{1}{2} \sum_i q_i \Phi_i \tag{3.7}$$

The main problem of our BD-step is to solve a sum-product in a massive parallel way because it is quite often used in BD-algorithm and builds up a

bottleneck for calculation performance. This is why we define the following general sum-product problem to be solved efficiently:

$$\sum_i q_i \Phi_i \tag{3.8}$$

All other calculations are not very expensive (depending on encounter-complex formulation). This is why we will evaluate various strategies for solving the sum-product problem on a GPU.

3.1.2 Trilinear Interpolation

We have got a discrete 3D grid with the potential $\Phi(r)$ for every grid point which have equal distances between 0.4 - 1.0 $\mathring{A}$ to each other. This potential grid is calculated by APBS which solves the Poisson-Boltzmann-Equation linearly (Section 2.3). The atom-coordinates of the protein are mostly between grid-points of our 3D grid and that is why we have to approximate the potential $\Phi(r)$ of every atom by interpolating trilinearly.
Generally, trilinear interpolation calculates the value of a point (sample point) which is based on the distance-weighted contribution of its eight neighbouring points in a grid space (Figure 3.1). For calculating a value for this point, respectively an atom, we have to build an average in every direction. At first, we build up a distance-weighted average in y-direction (red squares; U00 between U010 and U000, U01 between U010 and U011, U10 between U110 and U100). Then a distance-weighted average is build up for each pair in z-direction (green triangle; U0 between U00 and U01, U1 between U10 and U11). Finally, a distance-weighted average for the x-pair (between U0 and U1) is calculated. We get the final sample point U (blue dot) which results into a trilinear approximation between its eight neighbours. Since CUDA 2.0 (beta) NVIDIA

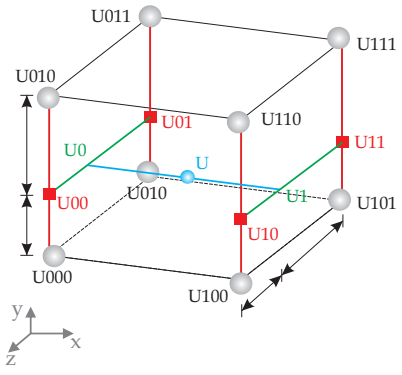


Figure 3.1: Trilinear interpolation in all directions in space

supports trilinear interpolation and has got a built-in function for this purpose if CUDA arrays are used (Section 2.6.4).

3.2 Parallelization Techniques and Strategies

At first, an introduction to an algorithm which is called Parallel prefix sum (Scan) is given and the code is modified to solve our sum-product problem defined in the last section. Afterwards it is shown how Parallel sum reduction and a matrix-vector multiplication can be performed efficiently on a GPU. These are important techniques for reducing bottleneck effects of the BD-algorithm with parallelization strategies.

3.2.1 Parallel Prefix Sum (Scan)

The Parallel prefix sum has got an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}] \tag{3.9}$$

and returns following result for an operator $\oplus$

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})] \tag{3.10}$$

For example: If we use + for the $\oplus$ operator and an array which looks like: $[1, 4, 2, 6, 4, 8]$, we get following result: $[1, 5, 7, 13, 17, 25]$.

This parallel algorithm is called inclusive scan. An exclusive scan is build by an inclusive scan by shifting all values by 1 and replacing the first element at position 0 with the identity of the $\oplus$ operator. A sequential exclusive scan which is executed on the CPU is trivial and is given in Algorithm 1.

Algorithm 1 void scan(float* inputdata, float* outputdata, unsigned int length)

```
1: outputdata[0] ← 0
2: for (j = 1; j < length; ++j) do
3:     outputdata[j] ← inputdata[j − 1] + outputdata[j − 1]
4: end for
```

This algorithm has a complexity of $O(n)$. Therefore, a work efficient parallel scan on the GPU should not increase this complexity to speed up the Scan algorithm compared to an implementation on the CPU. A work efficient implementation is realized by subdividing Scan into two parts which are called (1) up-sweep or reduce phase and (2) down-sweep phase (after Blelloch [Rei93], Figure 3.2).

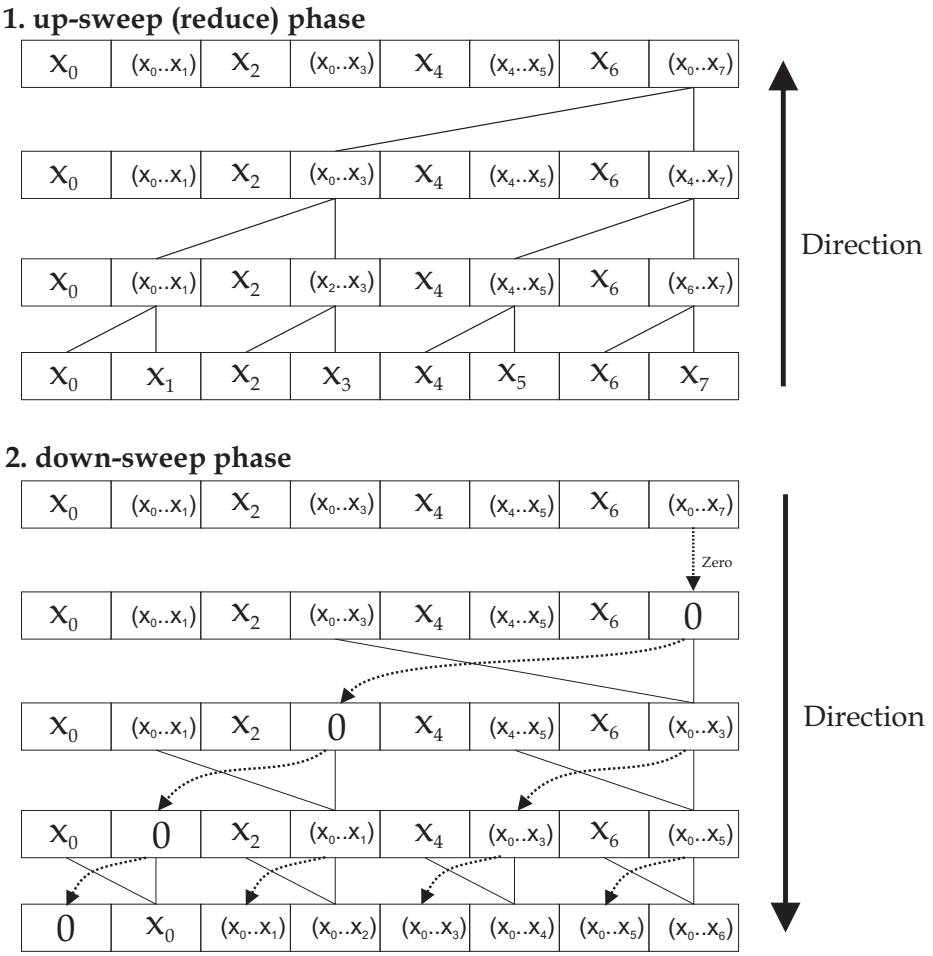


Figure 3.2: Up- and down-sweep phase of Parallel prefix sum (Scan)

For our purpose we need the up-sweep phase and the last element builds up the overall sum for all elements. To solve the whole equation 3.3 we have to multiply with appropriate q_i at the first step. A scheme for the whole modified algorithm is shown in figure 3.3.

Algorithm

The up-sweep phase (unmodified) is shown in algorithm 2 (down-sweep phase cf. [Har07]). At first, data is loaded in line 1 - 5 for every thread. Every thread holds a pair of elements to sum it up in the next step. Line 7 ensures that all threads have loaded their pairs until the first sum is made (line 11). The first condition in line 8 has to be fulfilled because every level has got half of sums to perform compared to one level before. If we are at the first level we have to perform half of the size of elements (in this example 8 / 2 = 4 sums). In the next level it is the half again (4 sums / 2 = 2 sums) until we are at the top

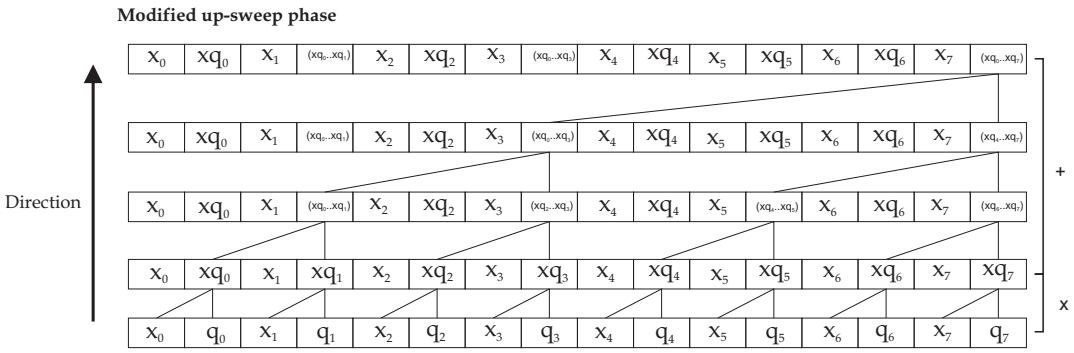


Figure 3.3: Up-sweep phase with multiplication at first step. A modification of Scan.

Algorithm 2 void prescan_reduction(float* inputdata, float* outputdata, unsigned int length)

```
1:  _shared_ float temp[]
2:  int thid ← threadID.x
3:  int offset ← 1
4:  temp[2 * thid] ← inputdata[2 * thid]
5:  temp[2 * thid] ← inputdata[2 * thid + 1]
6:  for (d = n>>1; d > 0; d >>= 1) do
7:     _syncthreads()
8:     if thid < d then
9:        int ai ← offset ·(2 · thid + 1) − 1
10:       int bi ← offset ·(2 · thid + 2) − 1
11:       temp[bi] ← temp[bi] + temp[ai]
12:    end if
13:    offset ← offset ·2
14: end for
```

level where we have to perform 1 sum and get the sum of all elements. In line 13 *offset* is doubled because after every step a sum has to be calculated for a pair of elements which has got a double distance to each other after every level.

Avoiding Bank Conflicts

With algorithm 2 we are getting bank conflicts. In fact, $offset = 1$ results to a 2-way bank conflict ($offset = 2$ results to a 4-way bank conflict). In figure 3.4 $thid$ and ai are calculated and arrows show which thread ($thid$) accesses which bank. For example $thid = 2$ calculates $ai = \text{offset} \cdot(2 \cdot thid + 1) - 1 = 2$ (offset = 1). Although $thid = 9$ calculates $ai = \text{offset} \cdot(2 \cdot thid + 1) - 1 = 18$ (offset = 1), $thid = 9$ has to access bank 2 because 16 banks form one block

and after the 16^{th} bank, bank 15, the access begins with bank 0 again.

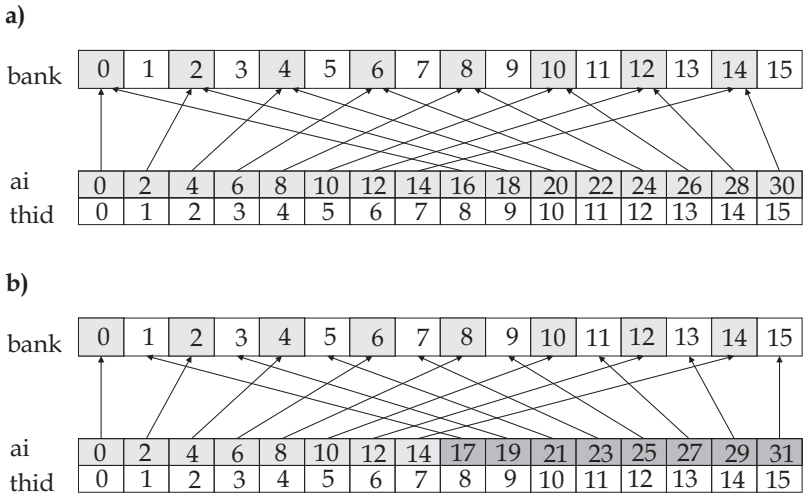


Figure 3.4: *a*: Addressing without padding: 2-way bank conflict, offset = 1, *b*: Addressing with padding: no bank conflict, offset = 1

To address with padding and therefore to avoid bank conflicts, algorithm 2 has to be enhanced by adding two lines (Algorithm 3.2.1) after line 10:

int $ai \leftarrow$ offset $\cdot(2 \cdot thid\ + 1) - 1$
int $bi \leftarrow$ offset $\cdot(2 \cdot thid\ + 2) - 1$
$ai \leftarrow ai\ +\ ai\ /$ Number of banks (16)
$bi \leftarrow bi\ +\ ai\ /$ Number of banks (16)
temp[bi] $\leftarrow$ temp[bi] + temp[ai]

CUDPP Implementation

An implemented version of Parallel prefix sum is available for CUDPP (Section 2.6.5). The algorithm is called by *cudppSegmentedScan*[1] where d_idata is the inputdata and d_out is the outputdata. Another built-in function which is called cudppMultiScan is interesting, too. With *cudppMultiScan* it is possible to perform a scan on multiple rows in parallel.

3.2.2 Parallel Sum Reduction

Parallel sum reduction is an algorithm which sums up a vector of values, respectively an array of words which can be linked by a $\oplus$ operator. It is a

[1]cudppSegmentedScan(CUDPPHandle planHandle, void * d_out, const void * d_idata, const unsigned int * _iflags, size _t numElements)

common and important data parallel primitive and has got a high parallelization potential. Generally the sum of values are built by many threads in blocks which are acting simultaneously on a portion of an array (Figure 3.5).

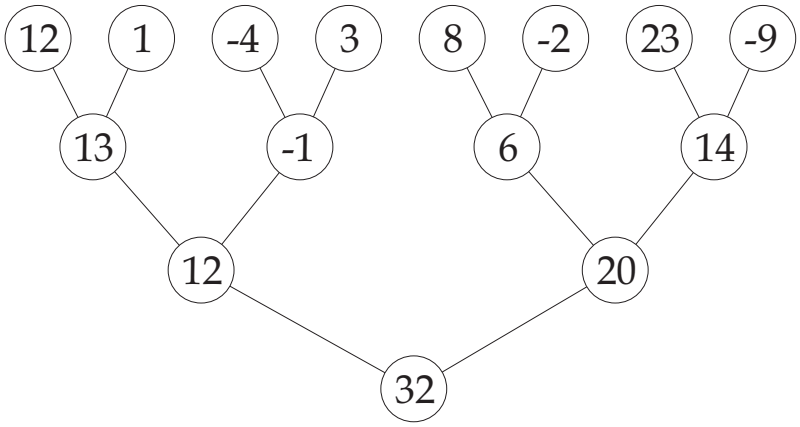


Figure 3.5: Tree-based approach for summation of an array

Every block of threads calculates one part of the whole array and writes its result back to an array which has to be summed again until there is one block left for calculation (Figure 3.6). Every kernel launch of this algorithm calculates a new level of our multiple tree which results into a decomposed computation. To know the possibility for optimization, an upper bound for

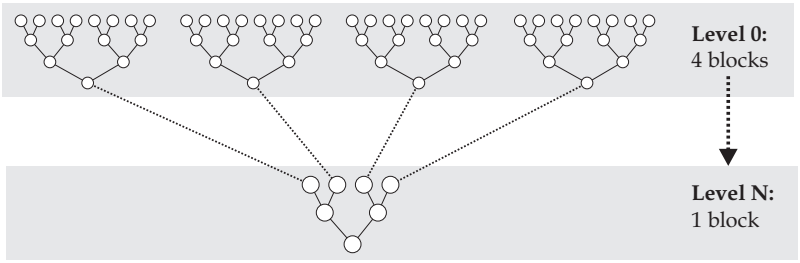


Figure 3.6: Decomposing computation into multiple kernel invocations

the bandwidth in GB/s (B) for a GPU is given by the equation 3.11.

$$B = \frac{\text{memory interface [bit]} \cdot 2 \cdot \text{clock rate [MHz]}}{\text{max. concurrent running blocks}} \tag{3.11}$$

For example a GTX 8800 has got following upper bound: $B = \frac{384 \text{ bit} \cdot 2 \cdot 900 \text{ MHz}}{8} \equiv 86.4\ GB/s$. Best performance ever reached by NVIDIA for this algorithm is a bandwidth of 73 GB/s on 32M elements. Also many optimizations have been made to reach this peak by reducing divergent branches of threads, avoiding bank conflicts, adding multiple elements per thread and completely unrolling all control structures with templates. A pseudocode is given in the next subsection with further explanation.

Algorithm

A pseudocode of the parallel sum reduction algorithm is given in Algorithm 3. It is separated into four parts. The first part (line 1 - 10) loads elements from global memory into shared memory for every thread. This is shown in figure 3.7. An array which is larger than the gridSize (= threads / block) is loaded by summation of all elements for every position i ($i \leq$ threads / block) from neighbour parts of the whole array. In the end the first part of the array has got a size of gridSize and a parallel summation is going to take place just in the first part because it has the overall sum of every part. Of course this can lead to worse performance if there is too much load from global memory, but we are able to choose the size of the array for reduction and call the reduction kernel until there is just one block left for processing. After loading all elements a

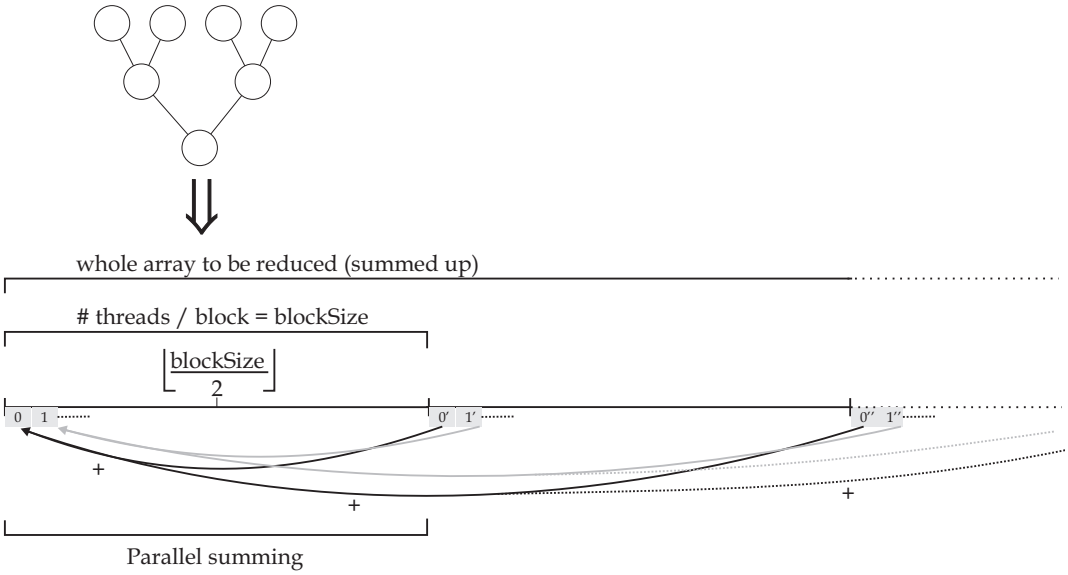


Figure 3.7: 1: Loading elements into first part of array

parallel summation has to take place in the first part of the array (line 11 - 16, Figure 3.8). To avoid bank conflicts every i^{th} thread accesses one block which has got his own elements indexed with $2^x + i$ (sequential addressing).

Lines 17 - 21 from Algorithm 3 shows how a loop unrolling (Section 2.6.4) is performed. If the number of threads is < 32 we are able to perform a complete loop unrolling by defining exactly which positions to sum up. This leads to a better performance because if we have got a low arithmetic intensity our bottleneck is likely to be the instruction overhead (address arithmetic and loop overhead) which has to be dissolved by compiler if not determined exactly by the programmer. That means, we do not need _syncthreads() and no if($tid < s$) because it does not save any work.

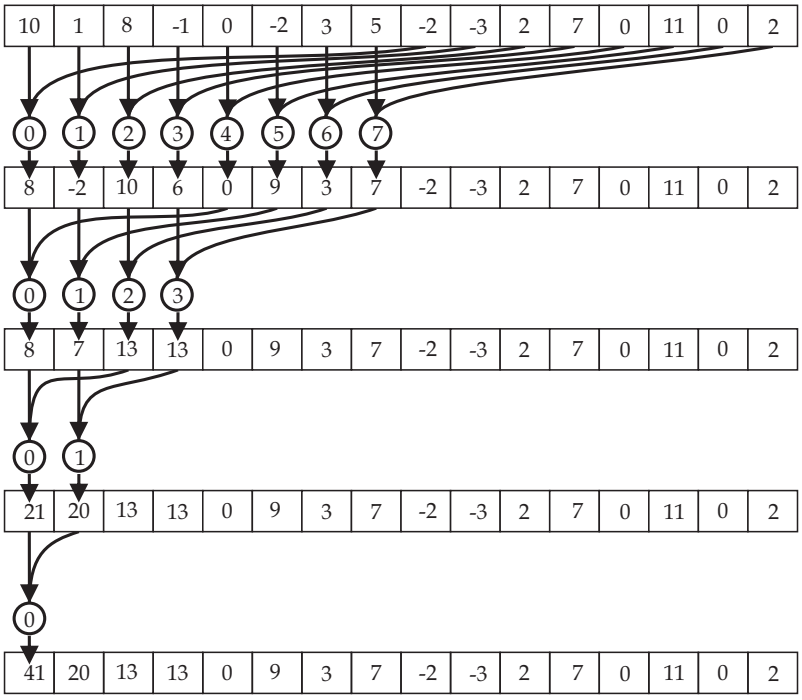


Figure 3.8: 2: Parallel summation of first part of array. Block processing with sequential addressing to avoid bank conflicts

The last three lines are for writing back the result in sdata[0] to outputdata in global memory for this specific block. Later on, we have to call *reduce(vars)* again until we have got a final sum (Figure 3.6).

Algorithm 3 void reduce(float* inputdata, float* outputdata, unsigned length)

```
1: _shared_ int sdata[]
2: int tid ← threadID.x
3: int i ← blockID.x · blockSize ·2 + tid
4: int gridSize ← blockSize ·2· gridDim.x
5: sdata[0] ← 0
6: while i < length do
7:    sdata[tid] ← sdata[tid]+ inputdata[i] + inputdata[i+blockSize]
8:    i ← i+ gridSize
9: end while
10: _syncthreads()
11: if blockSize ≥ 2^x (x ∈ ℕ, x ∈ [9..7]) then
12:    if tid < 2^(x−1) then
13:       sdata[tid] ← sdata[tid]+ sdata[tid + 2^(x−1)]
14:    end if
15:    _syncthreads()
16: end if
17: if tid < 32 then
18:    if blockSize ≥ 2^x (x ∈ ℕ, x ∈ [6..1]) then
19:       sdata[tid] ← sdata[tid]+ sdate[tid + 2^(x−1)]
20:    end if
21: end if
22: if tid == 0 then
23:    outputdata[blockID.x] = sdata[0]
24: end if
```

To use this algorithm for a BD-step it is possible to pre-calculate all $\Phi(r)$, store them in a linear array and before summation of neighbouring elements, each of them is multiplied with the corresponding q_i and we get an array which contains $\Phi_0 q_0, \Phi_1 q_1, \ldots, \Phi_{n-1} q_{n-1}$. This allows us to calculate the overall sum.

3.2.3 Dense Matrix-Vector Multiplication

A dense matrix-vector multiplication arise in many problems needed to solve a wide range of issues. Hence, it is important to speed up such calculations which is defined in following equation:

$$A \cdot x = y \tag{3.12}$$

A is a matrix ($\mathbb{R}^{m \times n}$) and x is a vector ($\mathbb{R}^m$). Therefore, y is our resulting vector ($\mathbb{R}^n$) of a sum-product. Applied to the BD-algorithm problem A holds

Φ_i and x has all q_i. y is therefore the resulting force, torque for x-, y-, z-dimension and energy of the protein PII.

In CUDA there are two implementations solving equation 3.12. The first one is defined in NVIDIA's BLAS library CUBLAS (Section 2.6.5). It provides a function called *sgemv*. A faster matrix-vector multiplication with a speed up of maximum ≈ 15 compared to the CUBLAS' implementation has been developed by Norijuki Fujimoto [Fuj08].

We briefly want to describe how a matrix-vector multiplication is performed on a GPU. A naive CPU implementation for solving $Ax = y$ is given in algorithm 4.

Algorithm 4 void mv_cpu(float* y, float* A, float* x, int m, int n)

```
1: for (i = 0; i < 0; i++) do
2:     y[i] ← 0
3:     for (j = 0; j < n; j++) do
4:         y[i]+ = A[i · n + j] · x[j]
5:     end for
6: end for
```

In case, designing an algorithm for the GPU the code has to be optimized for GPU architecture. An efficient way for performing a matrix-vector multiplication is done in the following way:

- For every thread load a 16 x 16 submatrix $P[i][j]$. This is done by loading w times a 16 x 16 submatrix for one row of A.
- Calculate $P[i][j]$ in every thread separately but at the same time in parallel.
- Build up the sum $P[i][j]$ for every thread.
- Write every result into vector y.
- Repeat all steps while it is possible to calculate a new row in the column with a 16 x 16 submatrix.

A schematic representation is shown in figure 3.9 and a pseudocode is given in algorithm 5. This code is constrained to A needing dimensions m and n which have to be a multiple of 16. To solve this problem a more detailed implementation is given by using CUDA arrays to save elements in texture memory for caching issues and using float4 datatypes for coalesced access without bank conflicts by Norijuki Fujimoto [Fuj08]. q is kept in shared memory so it is important that q does not exceed available shared memory.

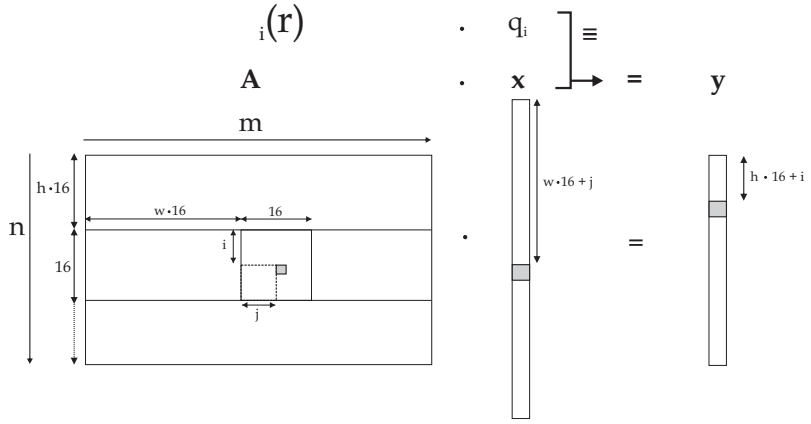


Figure 3.9: Performing a matrix-vector multiplication in parallel on the GPU

Algorithm

Line 1 of the parallel Dense matrix-vector multiplication algorithm (5) indicates that a sum has to be performed for rows in m-direction n/16 times, because the assumption, n being a multiple of 16, is been made for reason of clarity. $P[i][j]$ is initialized with 0 for every thread in line 2 and in lines 3 - 8 all $P[i][j]$ are calculated in m-direction m/16 times. The last lines (9 - 11) are calculation a total sum of $P[i][*]$ and assigning the result to vector y.

Algorithm 5 void mv_gpu(float* y, float* A, float* x, int m, int n)

1: **for** all h $(0 \leq h < n/16)$ in parallel **do**
2: float $P[16][16] \leftarrow 0$
3: **for** all $i,\ j$ $(0 \leq i, j < 16)$ in parallel **do**
4: $P[i][j] \leftarrow 0$
5: **for** $(w = 0;\ w < m/16)$ in parallel **do**
6: $P[i][j] \leftarrow P[i][j] + A[(16 \cdot h + i) \cdot n + (16 \cdot w + j)] \cdot x[16 \cdot w + j]$
7: **end for**
8: **end for**
9: **for** all i $(0 <= i < 16)$ in parallel **do**
10: $y[16 \cdot h + i] =$ total sum of $P[i][*]$
11: **end for**
12: **end for**

Chapter 4

Results

In this section a performance analysis for various parallelization algorithms (described in section 3) is given. At first, we show which testsystem is used to obtain results. Later on, a benchmark is presented which is given by the "Simulation of Diffusional Association" (*SDA*, Section 2.5.1). After this, the performance of Parallel prefix sum, Parallel sum reduction and matrix vector multiplications like Dense matrix-vector multiplication and a CUBLAS approach are discussed. Also an algorithm for the trilinear interpolation is implemented and a performance analysis is perfomed. All tests are repeated 10 times and we have drafted the average of all independent trials per measurement. Deviation between trials is very low, so there are no error bars indicating the error bounds in the following results. A comparison between all results is made in the discussion (Section 5) and results are summarized in figure 5.1.

4.1 Test Environment

The test environment performs sum-product calculations of the BD-algorithm. For analyzation and discussion, trends are recognized and a performance is measured compared to similar issues performing a fast sum-product. So it is not necessary to use best hardware to find out performance trends for various algorithms. Generally, all algorithms perform faster on Tesla products (by leastwise a factor of 2), but for our purpose we want to analyze trends. Efficiently programmed algorithms will always be scaled up by better hardware with compute device 1.x .
For this thesis all tests are performed on following host and device.

Host:

- Intel Core 2 Duo, Q6600, which has got 4 cores with 2,4 GHz for each one. On the host-side a calculation is always performed on one core each.
- 3,25 GB RAM (DDR-800, 2 x 400 MHz for each RAM).

Device:

- NVIDIA GeForce 8800 GT
- Revision number: 1.1
- 1 GB global memory space
- Multiprocessors: 14
- Number of cores: 112
- Maximal threads per block: 512

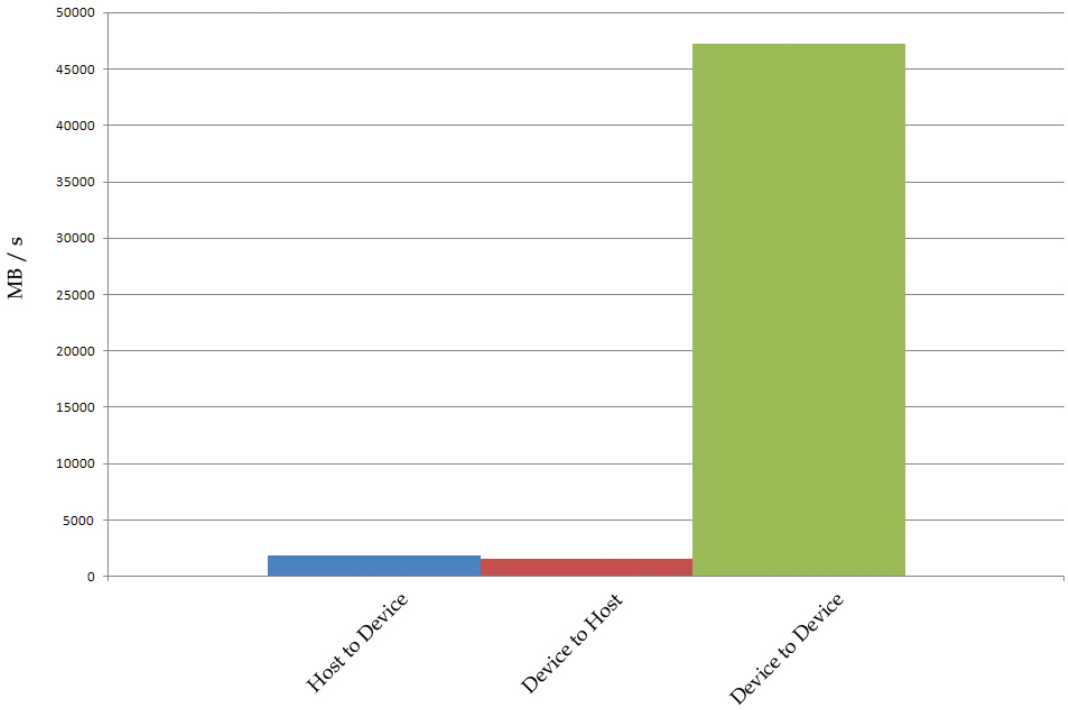


Figure 4.1: Bandwidth for pageable memory (GPU: GT8800, CPU: DDR 800, Q6600)

A data-transfer between the *host to device* (1,8 GB/s) is nearly as high as from *device to host* (1,5 GB/s) but a transfer inside the device has got an

average of 47,2 GB/s (Figure 4.1). The x-axis denotes the transfer direction and the y-axis shows a bandwidth in GB/s.

The theoretical bandwidth in GB/s for a 8800 GT (Equation 3.11) is given by $B = \frac{\text{memory interface [bit]}\cdot 2\cdot\text{clock rate [MHz]}}{\text{max. concurrent running blocks}} = \frac{256\text{[bit]}\cdot 2\cdot\text{clock rate [MHz]}}{8} \equiv 57.6GB/s.$ The highest peak we have reached is 47.2 GB/s (Figure 4.1). The difference 10 GB/s typically results from managing threads and data transfer access and other software specific issues.

4.2 Linear sum-product Calculation

To get to know how fast an efficient calculation of forces, torques or energies can be performed, a performance test of SDA is made for the force calculation part. This test is shown in figure 4.2. The ordinate defines the time in *ms* which is needed to perform a calculation for x atoms (axis of abscissa). A

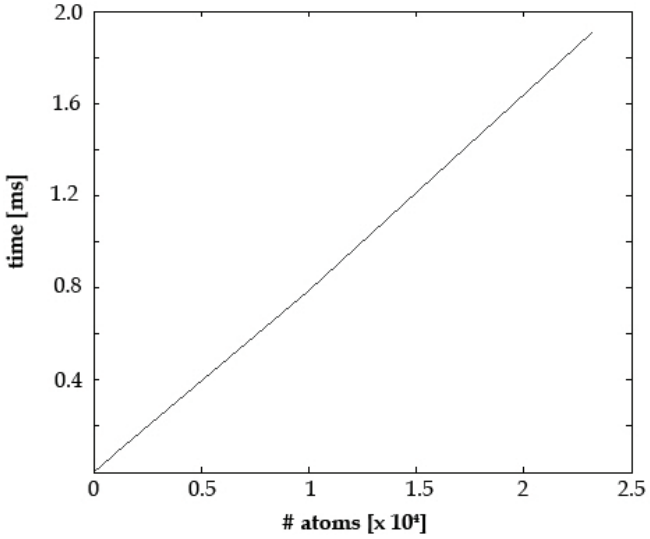


Figure 4.2: Performance for SDA in FORTRAN code running on one core at 2.4 GHz

nearly linear dependency is given for the atom-size and the time it takes for calculation. This result is to be expected because for every atom, we have to do a constant number (k) of calculations. The sum of all calculations is higher if the atom-size (n) is high. Therefore, we get a complexity of $O(k \cdot n) = O(n)$, respectively.

4.3 Parallel Prefix Sum Performance

In this algorithm we have two parts to consider. The up-sweep and the down-sweep phase (Figure 3.2). In both phases we have to start (resp., end) with calculating $n/2$ elements, then $n/4$ elements and so on. This results into a complexity of nearly $O(2 \cdot log(n)) \Rightarrow \ O(log(n))$. This behaviour is also measured in figure 4.3.

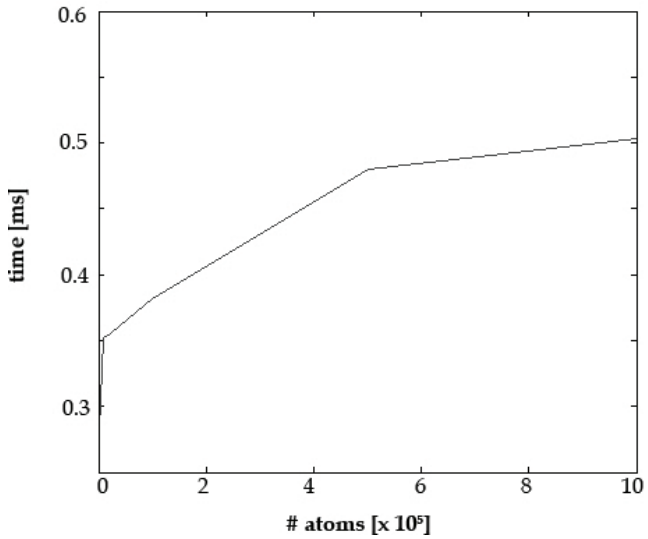


Figure 4.3: Parallel prefix sum performance

Figure 4.3 seems to look like $f(x) = log(x)$ and is compressed by a factor greater than 2 if calculation is reduced to a up-sweep phase.

Interestingly, in figure 3.3, we show, how to get a product into a sum at the first step. Because this is a switch in operation from $\cdot$ to $+$ the complexity does not change but the array has to be extended by q_i. q_i has to be copied into the array several times which doubles the size needed for the linear array. For our purpose all results are given after the up-sweep phase (which is similar the *Parallel sum reduction*).

Another approach is that every vector is written linearly into the array. Results are able to be fetched after the up-sweep phase, because a calculation of the overall sum for every vector is possible through subtraction of the prior one.

4.4 Parallel Sum Reduction Performance

The Parallel sum reduction calculates balanced-trees in blocks and writes its sum into an array which is also summed up until one block is left and we get the

overall sum of the array (3.5). Because every tree performs $log(n)$ operations, where n denotes the number of knots and leafs of one tree, and this has to be calculated k times it leads to a complexity of $O(log(n))$. We get the following performance measure (Figure 4.4):

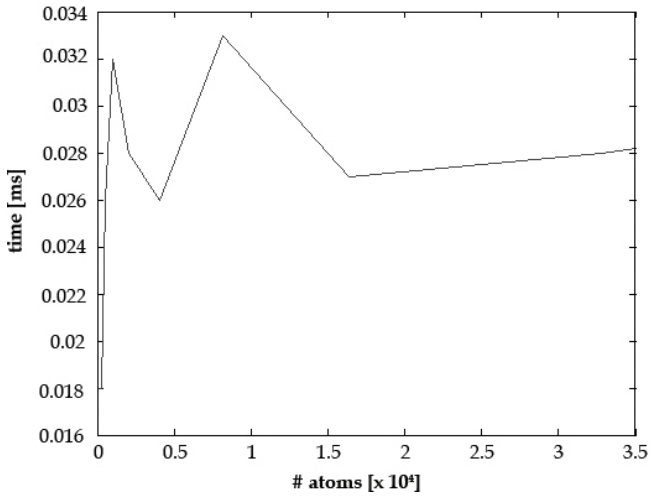


Figure 4.4: Parallel sum reduction performance

A tendency to $f(x) = log(x)$ is apparent but the results converge into a fluctuating range. This characteristic is observed because the algorithm is able to distribute the load-write balance in a more or less precise way. We are able to hide load-write operations which are expensive if a lot of threads are working concurrently. If this is the case the bandwidth raises up and the time for a summing up an array is able to fall although the array is larger than a smaller one. The size of an array has always got to be a multiple of 2 to get best performance out of the architecture for compute device 1.x .

For example in table 4.1 the average time for one BD-step calculating the overall force for 10000 atoms takes 0.054 ms and has got a bandwidth of 1.21 GB/s. The summing of 23150 atoms takes fewer time (0.028 ms) because it is possible to raise up the bandwidth up to 4.64 GB/s because of for example coalesced access of warps which leads to a higher occupation (Section 2.6.4).

The bandwidth used, raises up dramatically by increased array size (Figure 4.5). This is important because more blocks are calculated concurrently and if bandwidth would not increase the thread scheduler would have to work harder to get independent access to hardware resources. This leads to a worse latency hiding and therefore to higher overall execution time.

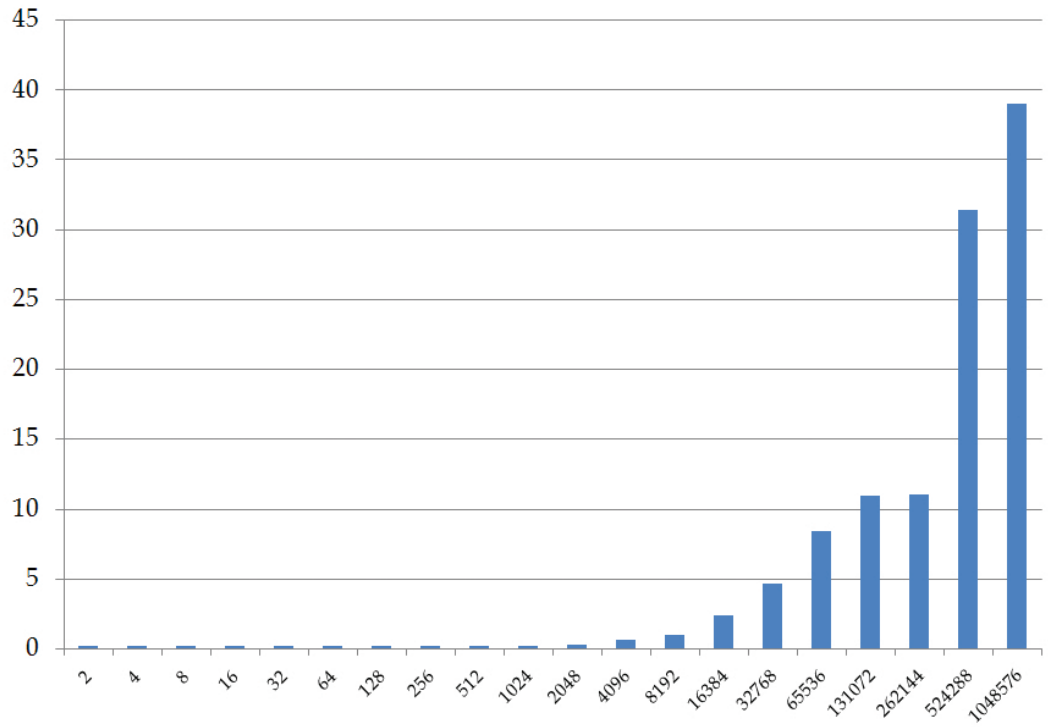


Figure 4.5: Bandwidth used for Parallel sum reduction

4.5 Matrix-Vector Multiplication Performance

Two different approaches solving a problem which is defined as $A \cdot x = y$ are evaluated and discussed to fit into our problem in the next subsections.

4.5.1 Dense Matrix-Vector Multiplication Performance

Performing a Dense matrix-vector multiplication (DMV) is fast for proteins which have thousands of atoms (# columns of A, Figure 3.9) and if matrix A has got a lot of rows (every row holds one direction of space for one protein). To get more than 3 rows which is needed if calculated one protein (forces of x-, y-, z-direction) it is also possible to perform a BD-simulation for two proteins many times in parallel. A visualization of the calculation is given by figure 3.9.

The complexity of the algorithm increases nearly linear because every thread calculates one "row" (16 x 16 matrix or more precise in a faster implementation a 16 x 64 array for A). A has got dimensions m and n. So the time for calculating m elements in parallel has to be done n times which has

# atoms	average time for one step [ms]	bandwidth [GB/s]
1	0.0176	0.0004
42	0.031	0.008
200	0.018	0.056
1000	0.027	0.157
10000	0.054	1.21
23150	0.028	4.64
50000	0.031	8.41
75000	0.036	10.9
100000	0.036	10.8
500000	0.066	31.4
1000000	0.107	39.01

Table 4.1: Dependency between bandwidth and GPU execution time

a linear cost if memory access is coalesced and there are no bank conflicts like it is fulfilled by *DMV*. For small atom-sizes the performance is bad because every time the algorithm performs memory has to be copied to a 2D CUDA array which is bind to a texture to get fast access to the array. As mentioned before the texture cache is 64kB and has got 8 kB cache included which makes fetching memory fast if an element is cached. A high speed up is reached if the m and n increase and therefore this method is much faster than calculating a sum-product linearly (Section 4.5.2).

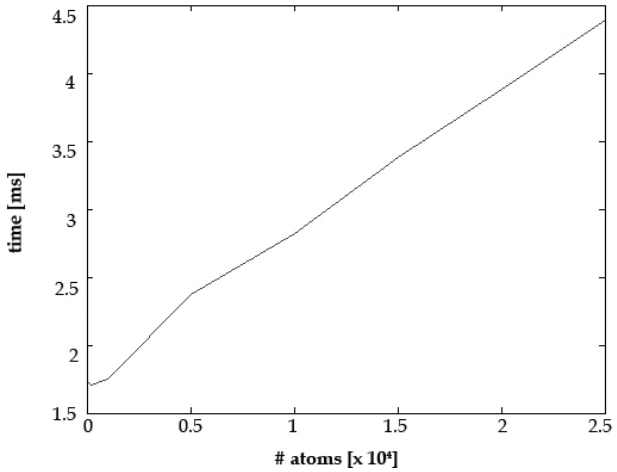


Figure 4.6: Dense matrix-vector multiplication performance for a single protein

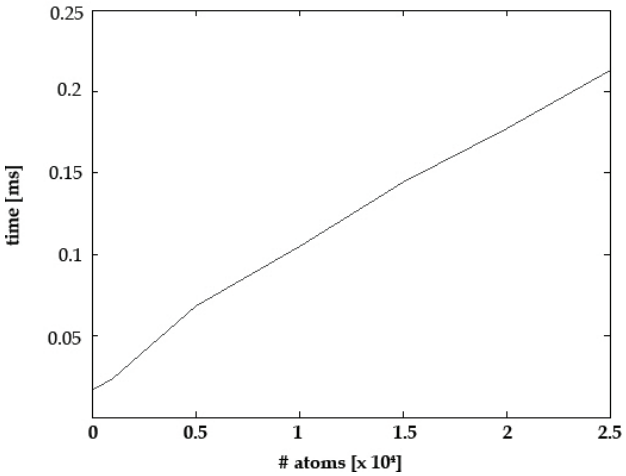


Figure 4.7: Dense matrix-vector multiplication performance for 100x concurrent BD-simulations divided by 100 for a realistic value of one BD-step calculation of one protein

To indicate the efficiency of many concurrent BD-simulation calculations figure 4.7 shows the performance. For example if 100 proteins with 10000 atoms need $\approx$ 1 ms to be calculated by Dense matrix-vector multiplication. Calculation of, e.g., forces for one protein with 1000 atoms by the linear sum-product algorithm needs 0.8 ms. This results to a boost of $0.8ms/1ms * 100 = 80$ (without interpolation).

4.5.2 Matrix-Vector Multiplication performed with CUBLAS

Figure 4.8 indicates the matrix-vector multiplication calculated with *sgemv* which is a part of the CUBLAS library v1.1. As we can see, for small values *sgemv* performs good and is twice as fast as *DMV*. But for matrices with higher order its performance decreases rapidly.

A comparison between *DMV* and *sgemv* leads to a high performance of *DMV* (Figure 4.9). While for 200 atoms *sgemv* is faster than *DMV* there is a speedup of 21.3x for 10000 atoms which is similar to evaluated values by Norijuki Fujimoto [Fuj08].

4.6 Naive Parallel Interpolation Performance

To get all Φ_i into A we have to make a preprocessing. We fetch one atom with x-, y-, z-elements, make a trilinear interpolation as described in 3.1.2 and have

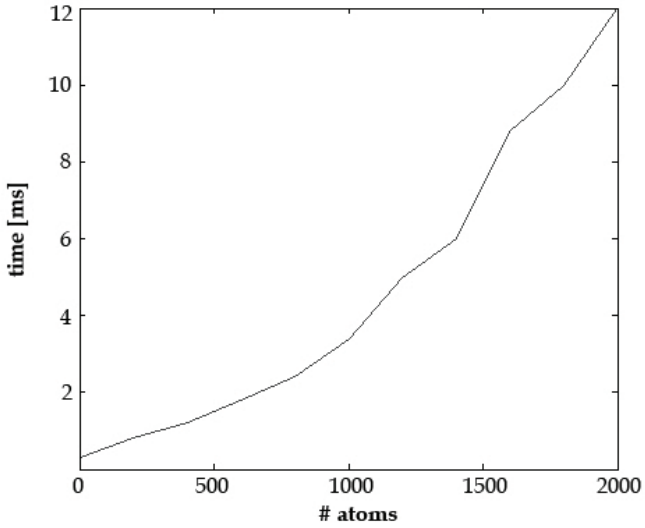


Figure 4.8: CUBLAS v1.1: *sgemv* performance for matrix vector multiplication

to write all results into A with few-way bank conflicts and coalesced memory access to get a high memory bandwidth. Performance is given in figure 4.10.

While the curve rises up until 5000 atoms it drops permanently again. This is the case because a decision is made of how many threads, blocks and grids are launched. The number of concurrent threads depends on the number of atoms to be calculated. We think that the number of bank conflicts rises cyclic dependent on the number of atoms compared to threads. This is why the Naive parallel interpolation has got potential to calculate much faster if decisions for using hardware parts are made dynamically.

In the next compute capability ($\geq$ 1.2) NVIDIA promises to provide a 3D-CUDA array to use texture memory and interpolate trilinearly between 8 neighbours. While this thesis this functions seems to be buggy and has got a driver conflict which should be fixed soon. That is why we separated this preprocessing from the real sum-product problem and think that this operation will soon be very efficient because it is demanded by the market.

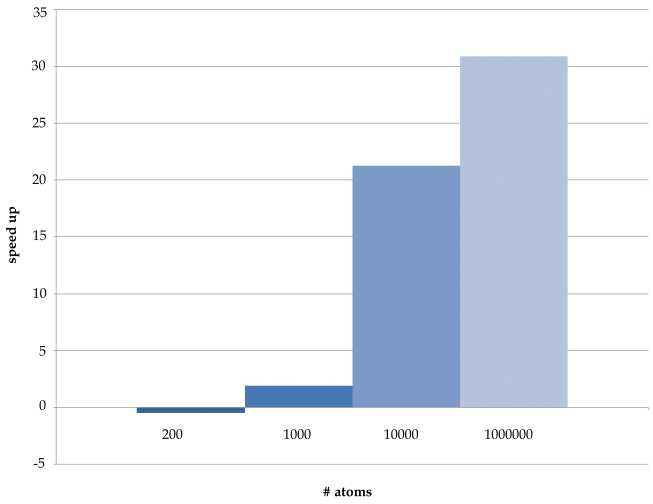


Figure 4.9: Speed up of Dense matrix-vector multiplication for one concurrent protein compared to *sgemv*

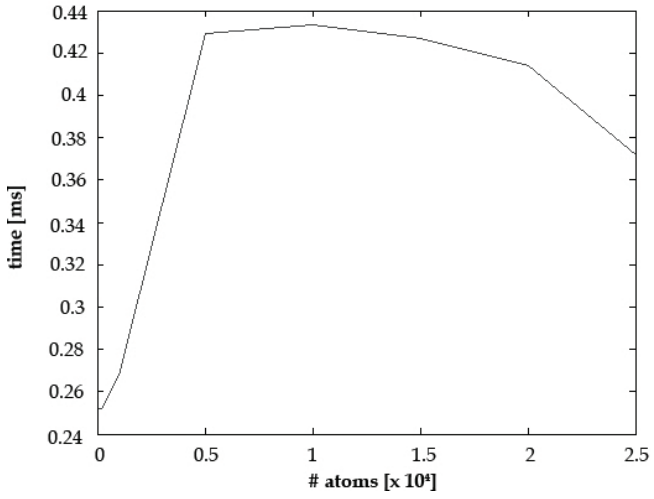


Figure 4.10: Naive parallel interpolation performance

Chapter 5

Discussion

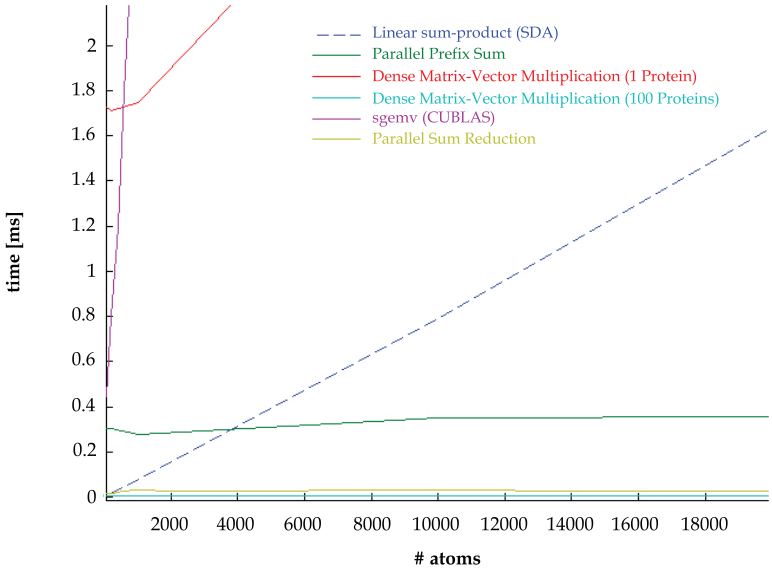


Figure 5.1: Comparison between all parallel algorithms (without preprocessing)

BD-simulations are important for calculation of association rates between proteins if they are diffusion limited. Understanding protein-protein interactions is crucial for simulating many biological mechanisms but calculation of protein-protein interactions in physical environment is computationally hard. Therefore, we need to make some simplification in theory. Simplification is made by simulation of two rigid proteins where one protein (PII) moves and rotates relative to another protein (PI). In this thesis we described how to calculate a force (resp., torque or energy) by summing all partial forces (resp., torque or energy) of every atom of PII which then moves and rotates in a well defined surface relative to PI after every step Δt. A solution for a general

problem which we denote as the sum-product problem

$$\sum_i q_i \Phi_i$$

has to be solved in a very efficient way. Several strategies have been shown in this thesis by using the G80 GeForce architecture (compute device 1.x) of NVIDIA.

A comparison between all parallel algorithms without preprocessing is given in figure 5.1. The Dense matrix-vector multiplication is 100 times higher than in the figure but because we want to compare one protein each this value is divided by 100 and we have to keep in mind that it indicates 100 BD-simulations in parallel. We will briefly discuss all parallel algorithms which solve the sum-product problem.

One strategy which has been evaluated is the upper-sweep phase of the Parallel prefix sum algorithm which is called Scan (Figure 5.2, x denotes Φ_i). At first, we have to perform a preprocessing by filling a linear array with $\Phi_i \cdot q_i$. After preprocessing we are able to calculate the sum (grey box) via Scan (possibility a). Another approach (possibility b) is to write a linear array with elements $\Phi_0, q_0, \Phi_1, q_1, \ldots, \Phi_{n-1}, q_{n-1}$ and to perform a multiplication at the first step and then an addition for every further step.

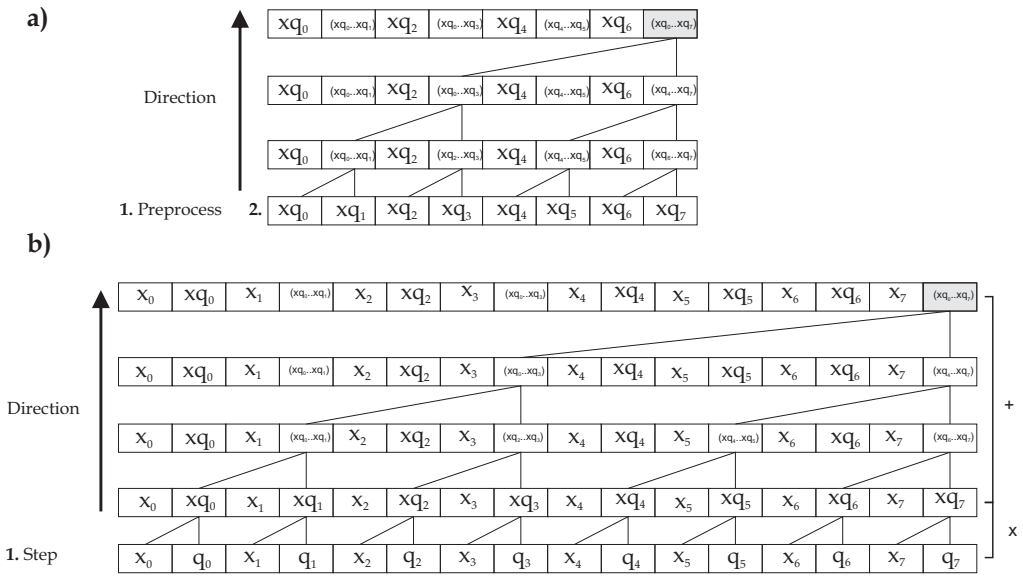


Figure 5.2: Upper sweep-phase of Scan. Two possible strategies for solving the sum-product problem. x denotes Φ_i.

The complexity is $O(log(n))$ but to be faster than a linear sum-product calculation the size of the linear array has to be huge to compensate time

for data-loading from host to device, execution time for threads and other mechanisms to perform efficient usage of GPU elements (details are listed in section 2.6.2). This means that for small arrays a GPU cannot be faster than a linear sum-product calculation on the CPU. Given the obtained results we need ≥ 3800 atoms which have to be processed to be faster with parallel Scan than the linear sum-product algorithm (Section 4.2). But if this requirement is fulfilled we will get a speed up of 4.3 for example for 20000 atoms (time for linear sum-product: 1.6 ms, time for Scan: 0.37 ms), if preprocessing has a cost of 0, therefore speed up is smaller in reality.

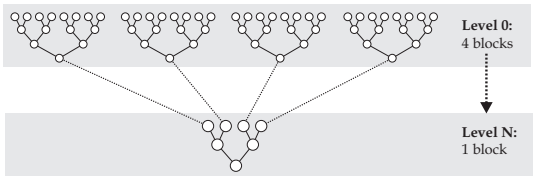


Figure 5.3: Parallel sum reduction scheme

Another approach has been given by the Parallel sum reduction algorithm (Reduction, Figure 5.3). This algorithm performs a calculation of balanced trees by summing up all values in $O(log(n))$. This summation is performed in parallel for many balanced trees. A partial sum is build up and is reduced again until one block is left for calculation with a last balanced tree to get the overall sum (Level N). The smallest performance we get is 0.0176 ms for 1 atom. 0.0176 ms are reached by the linear sum-product if ≈ 220 atoms are processed. A size between $220 \leq 256$ for Reduction is as fast as the linear sum-product approach. But a size of 1024 performs to a speed up of 30 (time for linear sum-product: 0.82 ms, time for Reduction: 0.027) which is faster than the Scan approach if trilinear interpolation has no cost. Because of an atom has got a x-, y- and z-coordinate and we have to calculate a force in every direction the speed up is reduced to a factor of 10 because we have to perform Reduction 3 times for one atom. Because of balanced-tree summing which is processed always in one block the factor is mostly higher in reality.

A third way for solving the sum-product problem is given by solving $A \cdot x = y$ (Section 3.2.3 and 4.5) , where A denotes a matrix (with dimensions $m \cdot n$) and x, y are vectors. A stores all Φ_i, x stores q_i and y is our result with all summed up results needed to perform a BD-step (Figure 5.4). We performed two ways of solving $A \cdot x = y$. The first approach is called Dense matrix-vector multiplication and the second one is an implementation of a matrix-vector multiplication called *sgemv* in CUBLAS (v1.1, Section 2.6.5). Our measured data shows that the Dense matrix-vector multiplication is at least as fast as *sgemv* but speeds up up to ≈ 30 for larger matrices A. Therefore, Dense matrix-vector multiplication is chosen to boost our BD-algorithm.

First of all, at least ≈ 1.7 ms are needed for setting up all arrays and

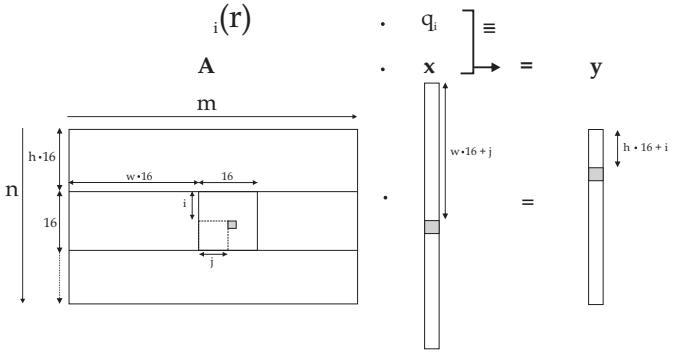


Figure 5.4: Performing a matrix-vector multiplication in parallel on the GPU

datatypes for Dense matrix-vector multiplication. This means, we need at least $\approx$ 5000 atoms (not shown in figure 5.1) to be as fast as the linear sum-product algorithm if we have got 3 rows for matrix A.

A great performance increase is reached, if several BD-simulations are performed simultaneously. In this case we could perform, e.g., 100 BD-simulations in parallel with the same two proteins and compare diffusion association rates with each other. We have measured such a setup in figure 4.7.

For example 100 proteins with 1000 atoms (nearly same time for 10000 atoms) need $\approx$ 1 ms to be calculated by the Dense matrix-vector multiplication algorithm. Calculation of, e.g., forces for one protein with 1000 atoms needs 0.8 ms for the linear sum-product algorithm. Although 0.8 ms is smaller than 1 ms we performed 100x a BD-simulation with the Dense matrix-vector multiplication instead of one BD-simulation with the linear algorithm. This results in a boost of $0.8ms/1ms * 100 = 80$ (without preprocessing). This speed up rises if more BD-simulations are performed simultaneously.

In fact, we want to calculate different proteins and would like to simulate for example not 100x the same protein but 100 different proteins. This is not really possible because $x \equiv q_i$ has to be loaded into shared memory (16kB * 1024 / 4 Bytes = 4096 floats to be stored maximal without control variable consideration and other important constants which have to be declared in shared memory) and every protein needs other effective charges q. But if it is possible to load all different q into shared memory, this could also be possible to consider in near future.

For preprocessing purposes a trilinear interpolation has to be performed and added to overall BD-simulation performance. We implemented a naive parallel trilinear interpolation algorithm which in fact has got optimization potential (Section 4.6). Also a new function for allocation of 3D-CUDA arrays is going to be provided by NVIDIA which has the ability to perform an interpolation in a 3D grid-space by trilinear interpolation of a point with its 8 neighbours (Section 3.1.2).

Chapter 6

Conclusion

In this thesis the aim is to understand the main routines, i.e. force and energy calculation for BD-simulation and to effectively parallelize them on NVIDIA GPU hardware. Thereby, the development, implementation and evaluation of different parallelization strategies are performed.

In summary, we get an appropriate speed up by a GPU which can be a factor of 80 and more. It is highly dependent on the problem we want to solve. A sum-product of a huge array (≥ 1024 atoms) is performed by Parallel sum reduction algorithm with a speed up of nearly 30. A speed up of ≥ 80 is reached by a Dense matrix-vector multiplication for calculation of multiple BD-simulations on same two proteins. A Parallel prefix scan algorithm also performs good results for arrays which are ≥ 3800 atoms but is not as fast as the Parallel sum reduction algorithm.

In future, internal forces will have to be calculated for modelling protein flexibility which has got an important effect on protein-protein interactions. Also a multi protein BD-simulation should be performed to understand dynamics between multiple protein interactions. The most important step is to design an algorithm which is highly parallelizable and therefore will scale up with future hardware improvements while dynamic allocation of memory usage is made by smart programming. It is expected to reach a higher performance than 1 Teraflop of calculation speed this year by NVIDIA's G200 GPU and a doubling of performance has been announced every two years. Another important developments are being made for example by INTEL. They are scaling up multi-core processor systems and also working on systems called "Larabee" which will have ≈ 32 cores.

If the BD-algorithm is designed as described in section 2.6.6 such as memory bank conflicts are avoided, coalesced memory access is used and a huge number of threads and blocks are launched with high occupancy and also shared memory of cached memory is used in a massive parallel way, a speed up of, e.g., 30 could be performed with the Parallel sum reduction algorithm

to perform an efficient and fast BD-simulation or many BD-simulations concurrently which leads to a boost of ≥ 80 or more by upscaling or using better hardware in the near future, respectively.

Chapter 7

Appendix A

7.1 Technical Details on G80

	abbreviation	quantity
Multiprocessors per GPU	**mpg**	16
Threads per warp	**tpw**	32
Warps per multiprocessor	**wpm**	24
Threads per multiprocessor	**tpm**	768
Blocks per multiprocessor	**bpm**	8
Registers (32-bit) per multiprocessor	**rpm**	8192
Shared memory (bytes) per multiprocessor	**smpm**	16384

Table 7.1: Architectural details on G80 (GTX)

7.2 Occupancy Calculation

Note that $ceil(x, y)$ indicates the next multiple of y where $c \cdot y \geq x$ and $c \in \mathbb{N}, c \geq 1$. Also all abbreviations are given in Appendix 7.1. We are free to set dimensions for a kernel launch with tpb_v (threads per block), rpt_v (registers per thread) and $smpb_v$ (shared memory per block). We have to design our algorithm the way trying to $max \Leftarrow$ (occupancy per multiprocessor) (Equation 7.11).

7.2.1 Processing per Block

$$\text{wpb}_* = ceil\left(\frac{tpb_v}{\mathbf{tpw}}, 1\right) = ceil\left(\frac{tpb_v}{32}, 1\right) \tag{7.1}$$

$$\mathrm{rpb}_* = \mathit{ceil}\left(\mathrm{wpb}_* \cdot 2,\ 4\right) \cdot 16 \cdot \mathit{rpt}_v \tag{7.2}$$

$$\mathrm{smpb}_* = \mathit{ceil}\left(\mathit{smpb}_v, 512\right) \tag{7.3}$$

7.2.2 Maximum Blocks per Multiprocessor

$$\textit{limited by } \mathrm{wpm}_* := \mathit{min}\left(\mathbf{bpm}, \mathit{ceil}\left(\frac{\mathbf{wpm}}{\mathrm{wpb}_*},\ 1\right)\right) \tag{7.4}$$

$$\textit{limited by } \mathrm{rpm}_* = \begin{cases} \mathit{ceil}\left(\frac{\mathbf{rpm}}{\mathrm{rpb}_*},\ 1\right) & \text{if } \mathit{rpt}_v > 0, \\ \mathbf{bpm} & \text{other} \end{cases} \tag{7.5}$$

$$\textit{limited by } \mathrm{smpm}_* = \begin{cases} \mathit{ceil}\left(\frac{\mathbf{smpm}}{\mathrm{smpm}_*},\ 1\right) & \text{if } \mathit{smpb}_v > 0, \\ \mathbf{bpm} & \text{other} \end{cases} \tag{7.6}$$

7.2.3 GPU Occupancy

$$\text{active blocks per multiprocessor} := \mathrm{abpm}_* = \mathit{min}\left(\mathrm{wpm}_*,\ \mathrm{rpm}_*,\ \mathrm{smpm}_*\right) \tag{7.7}$$

$$\text{active warps per multiprocessor} := \mathrm{awpm}_* = \mathrm{abpm}_* \cdot \mathrm{wpb}_* \tag{7.8}$$

$$\text{active threads per multiprocessor} := \mathrm{atpm}_* = \mathrm{abpm}_* \cdot \mathbf{tpb} \tag{7.9}$$

$$\text{max. simultaneous blocks per multiprocessor} := \mathrm{abpm}_* \cdot \mathbf{mpg} \tag{7.10}$$

$$\textbf{occupancy per multiprocessor} := \frac{\mathrm{awpm}_*}{\mathbf{wpm}} \tag{7.11}$$

Bibliography

[BL92] Andrew McCammon Brock Luty, Malcolm Davis. Electrostatic energy calculations by a Finite-difference method: Rapid calculation of charge-solvent interaction energies. *Journal of Computational Chemistry*, 1992.

[Cor08] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture*, 2008. version 2.0.

[ea95] J. Madura; J. Briggs; R. Wade; et al. Electrostatics and Diffusion of Molecules in Solution: Simulations with the University of Houston Brownian Dynamics Program. *Comp. Phys. Commun.*, 91:57–95, 1995.

[EDM85] Stuart A. Allison Eric Dickinson and J. Andrew McCammon. Brownian Dynamics with Rotation-Translation Coupling. *J. Chem. Soc*, 81:591 – 601, 1985.

[Ein05] Albert Einstein. Investigations on the Theory of the Brownian Movement. *Ann. der Physik*, 1905.

[EM78] Donald L. Ermak and J. Andrew McCammon. Brownian dynamics with hydrodynamic interactions. *J. Chem. Phys.*, 69, 1978.

[Fuj08] Noriyuki Fujimoto. Faster Matrix-Vector Multiplication on GeForce 8800GTX. *IEEE International Parallel and Distributed Processing Symposium*, 2008.

[GW96] Razif R. Gabdoulline and Rebecca C. Wade. Effective Charges for Macromolecules in Solvent. *J. Phys. Chem.*, 10:3868–3878, 1996.

[GW98] Razif R. Gabdoulline and Rebecca C. Wade. Brownian Dynamics Simulation of Protein-Protein Diffusional Encounter. *Methods*, 1998.

[Har07] Mark Harris. Parallel Prefix Sum (Scan) with CUDA. Technical report, NVIDIA, 2007.

[Hel06] A. Spaar; C. Dammer; R. R. Gabdoulline; R. C. Wade; Volkhard Helms. Diffusional Encounter of Barnase and Barstar. *Biophysics Journal*, 10:1913–1924, 2006.

[Inca] NVIDIA Corporation Inc. CUDA Occupancy Calculator and other downloads. `http://www.nvidia.com/object/cuda_develop.html`.

[Incb] NVIDIA Corporation Inc. CUDA Showcase. `http://www.nvidia.com/object/cuda_home.html`.

[Lib] CUDA Data Parallel Primitives Library. GPGPU. `http://www.gpgpu.org/developer/cudpp/`.

[LNP07] Mark Harris Lars Nyland and Jan Prins. *GPU Gems 3: Chapter 31, Fast N-Body Simulation with CUDA*. Addison-Wesley, Boston, 2007.

[NB01] F. Wang N. Baker, M. Holst. Adaptive multilevel finite element solution of the Poisson-Boltzmann equation. *IBM Journal of Research and Development*, 2001.

[paL] Science-Based prediction at LANL. SciDAC. `http://www.scidacreview.org/0702/html/hardware.html`.

[Rei93] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[vS06] M. von Smoluchowski. Zur kinetischen Theorie der Brownschen Molekularbewegung und der Suspensionen. *Ann. der Physik*, 1906.

[ZtW05] Van Zon and P. ten Wolde. Simulating biochemical networks at the particle level and in time and space. *Physical Review Letters*, 2005.