Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3080

Numerical Accuracy Analysis Based on the Discrete Stochastic Arithmetic on Multiprocessor Platforms

Christian Mötzing

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Sven Simon
Supervisor:	M.Sc. Wenbin Li
Commenced:	August 02, 2010
Completed:	February 01, 2011
CR-Classification:	G.1.0, I.6.3, D.0

Abstract

Simulating the real world has become one of the most widely used techniques in engineering today. Multiprocessor platforms play a key role in this development since bigger and bigger problems need more computing power to be solved. When the floating point standard was adopted in the early eighties of the 20th century, the amount of floating point operations executed in a simulation was very low compared to today. Nowadays, numerical errors accumulate to a noticeable amount, what is known as *round-off error propagation* and describes the problem that this error can grow over time, finally making the result worthless in terms of informational content.

Where lives, money or other critical aspects depend on computed results confidence about their correctness is of paramount importance. Therefore numerical analysis techniques were developed to make a statement about the accuracy of results computed with floating point arithmetic. They are well defined and understood in the theoretical world but rarely implemented or used in applications. This thesis will develop an approach to implement the accuracy analysis Discrete Stochastic Arithmetic in software aiming at integrating into an existing software package for simulating molecular dynamics. Discrete Stochastic Arithmetic is based on CESTAC, one of the first methods used for estimating round-off errors. An emphasis will be laid on easy applicability and improved performance of the developed methods. To review the effectiveness of the implementations a case study will be performed on a simple simulation example.

General-Purpose computation on Graphics Processing Units (GPGPU) has recently established its reputation in scientific computing for accelerating parallelizable computations. Due to their completely different architecture, with hundreds of specialized cores, modern graphics cards achieve high ratings for floating-point operations per second (Flops). The difference to common CPU architectures also has a downside: developers need to rethink their usual implementation approaches and learn to handle the different tool and instruction set provided. This thesis will elaborate on the possibilities of implementing Discrete Stochastic Arithmetic on GPU as well as the merits compared to CPU.

Keywords: *Discrete Stochastic Arithmetic, Accuracy Analysis, Round-Off Error Propagation*

Acknowledgements

First of all I would like to thank my advisor M.Sc. Wenbin Li for all his patience and good advice throughout the development of this thesis.

Also I'd like to thank Prof. Dr. Sven Simon and the department for Parallel Systems for giving me the opportunity to work on an interesting problem.

My gratitude also goes to Prof. Dr. Jürgen Pleiss, Sascha Rehm and Tobias Kulschewski from the Institute of Technical Biochemistry at the University of Stuttgart for their invaluable advice and insights on molecular dynamics simulations.

Finally I deeply thank my parents for supporting me throughout my studies making my graduation possible.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This chapter gives an overview over the thesis. First a motivation to the problem is given in section 1.1 followed by the definition of goals of the thesis (section 1.2). Concluding there is an overview over the chapters (section 1.3).

1.1 Motivation

During the initial period of scientific computing use of limited accuracy floating point arithmetic as adopted by the IEEE 754[1] standard has never been challenged concerning round-off errors. Floating point operations per second (Flops) where limited by slow hardware and therefore everyone assumed that loss of information during arithmetic operations would not affect the overall result. Nowadays, a single GPU achieves about 3 TFlops[2]. For comparison: When the first TOP500[3] list of super computers was published in 1993, all 500 Systems had a combined computing power of 1.122 TFlops. In 1993 executing a simulation on the fastest system took 1 hour, on today's fastest system one can easily do the same 20 times within less than a second (if we are looking at Flops only). This demonstrates the rapid development of computing power in this area, which is surely good for scientific progress as models can be more complex and simulations more thorough.

Unfortunately, a growth of arithmetical operations with several orders of magnitude arises a problem. Using IEEE 754 as accurate binary floating-point representation will not always be sufficient any more. Billions of rounded operations produce what is known as *round-off error propagation*. This denotes the fact that rounding errors accumulate over time until they make up a significant part of the result. This makes results less trustworthy and can, in extreme cases, completely erase the informational statement. If lives, money or other critical aspects depend on such results it is not acceptable to just assume correctness. Rather absolute confidence concerning the accuracy of computational results is inevitable.

Mathematicians developed a way to estimate the round-off error based on a numerical analysis: **D**iscrete **S**tochastic **A**rithmetic (DSA). CADNA[4] is the first library to implement DSA but

[1] http://grouper.ieee.org/groups/754/
[2] http://www.amd.com/us/press-releases/Pages/amd-press-release-2009sep22.aspx
[3] http://www.top500.org/list/1993/06/100
[4] http://www-pequan.lip6.fr/cadna/

slows down calculations so much that it is hardly usable for long running simulations. Also it is not possible to use CADNA in software packages utilising graphics cards to accelerate floating-point calculation intense applications.

1.2 Goals

The goal of this thesis is to assess the applicability of numeric accuracy analysis based on the Discrete Stochastic Analysis. To do so, a way is needed to map the theoretical idea to a software solution. Important are the aspects of easy applicability and improved execution performance. To review the solution's applicability a case study is carried out on molecular dynamics simulation package GROMACS[5].

The second part will focus on implementing DSA for the **G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits[6] (GPGPU) platform. The biggest incentive is the possibility to achieve a further acceleration compared to CPU versions of accuracy analysis. Due to its totally different architecture porting a solution to GPU arises different problems. A NVIDIA based graphics card was made available and thus used in this thesis. There are a several programming environments available for NVIDIA GPUs like BrookGPU[7], OpenCL[8], Direct Compute[9] and **C**ompute **U**nified **D**evice **A**rchitecture[10] (CUDA) but since CUDA is closest to hardware for NVIDIA GPUs this thesis will implement a CUDA based solution.

1.3 Outline

The thesis has following structure. Chapter 2 explains the basic idea of the Discrete Stochastic Arithmetic. The IEEE standard for floating point arithmetic is introduced because it is key to understand the underlying problem. Section 2.2 gives a brief explanation of DSA and the theory behind it. The last section introduces the first library implementing an accuracy analysis method based on DSA. Chapter 3 discusses the different approaches of how to implement DSA in a software package and finally presents the strengths and weaknesses of each method. Chapter 4 is devoted to the software package GROMACS and the case study where implementing DSA is explained and the results of simulations are examined. The chapter finishes with a conclusion of the results obtained. Chapter 5 agitates the differences

[5] http://www.gromacs.org
[6] http://gpgpu.org/about
[7] http://graphics.stanford.edu/projects/brookgpu/
[8] http://www.khronos.org/opencl/
[9] http://www.nvidia.com/object/cuda_directcompute.html
[10] http://www.nvidia.com/object/what_is_cuda_new.html

between the GPU and the CPU architecture. It tries to clarify the benefits for scientific computing and introduces the DSA implementation for GPU.

2 Discrete Stochastic Arithmetic

The term Discrete Stochastic Arithmetic (DSA) was formed by J. Vignes and first mentioned in [Vig93]. It is based on a method for round-off error analysis named *Control et Estimation Stochastique des Arrondis de Calcul (CESTAC)* described in [PV80], [Vig88] and [Vig78]. Sometimes Permutation-Pertubation is mentioned as well, but since it contains additional approaches to the problem of round-off error propagation, only the first two will be referred to.

2.1 Standard for Binary Floating-Point Arithmetic (IEEE 754)

The IEEE 754 standard was adopted in 1985 and describes how binary floating-point numbers should be handled by microprocessors. There were four proposed formats. The most widely known are the *Single Format* which needs 32 bits and the *Double Format* which needs 64 bits to represent one number. Usually those two formats are offered to the user through the high-level languages. The other two formats, the *Single-Extended Format* ($\geq$ 43-bit) and the *Double-Extended Format* ($\geq$ 79-bit, usually 80 bits), are optional and do not have to be implemented to satisfy the standard. The same is true for *Double Format*.

The representation of a value in binary format is encoded in three parts: sign, exponent and mantissa. Table 2.1 lists how many bits are used for each of these three parts. The precision

Precision	**Sign**	**Exponent**	**Mantissa**
single	1 bit	8 bits	23 bits
double	1 bit	11 bits	52 bits

Table 2.1: Floating-point memory representation

of the representable numbers is limited by the length of the chosen format. The result of $1/3$ equals to $0.\overline{333}$ and no matter how high the chosen precision is, the exact result will be cut off eventually. To control this behaviour common to all mathematical operators the IEEE 754 standard defines several rounding modes.

Round to Nearest Returns the next representable value closest to the exact result. This is the default rounding mode.

Round Up Round toward $+\infty$

Round Down Round toward $-\infty$

Round toward Zero Returns the value closest to zero.

Note that rounding applied to a specific value will only result in two different results. For example round to zero equals round down for positive values. Accordingly round up and round to zero are equal for negative values.

The standard furthermore allows the microprocessors to internally work with a high precision format and then round the result to the precision used in the program (e.g. 32 bits or 64 bits). As a consequence, microprocessors are mostly using 80 bits internally to achieve higher precision results. Especially operations with intermediate results, like multiply-add-fused instructions, benefit from this implementation. Multiply-add-fused instructions multiply two floating-point operands and the result is added to the third operand without an intermediate rounding operation in-between. In terms of accuracy this certainly is desirable. However, in some cases it impedes the reproducibility of simulations, especially in case of dynamic load balancing being used to execute simulations on multiple processors, because associativity for mathematical operations is no longer given (see [CK]). Some of these aspects which have an influence on GROMACS are listed in [gro].

The ISO C99 standard provides compatibility with IEEE 754 and therefore offers functionality to control the [1]FPU. The interface allows to read from and write to different registers, interact with floating point exceptions and control the rounding mode. A detailed description can be found in [fen].

2.2 Fundamentals of Discrete Stochastic Arithmetic

Chapter 2.1 already gave an introduction to floating-point arithmetic implemented on microprocessors and why calculations do not yield a mathematic exact result. To get an idea of how to quantify the error, one has to look at the theoretical definition of floating-point arithmetic. Let Ω be the binary floating-point operator used by a microprocessor and ω the exact mathematical operator. A computation performed by a micro-processor can be described as

$$X\Omega Y = X\omega Y - 2^{E-p}\epsilon\alpha$$

[1]Floating Point Unit

where E is the binary exponent of the result, p the length of the mantissa in bits, ϵ the sign of $X\Omega Y$. Further, $2^{-p}\alpha$ represents the absolute round-off error (see [Che90]). Changing the rounding mode for each operation introduces a random variable h. Using round to nearest h is, for example, randomly equal to -1, 0, or $+1$.

$$X\Omega Y = X\omega Y - 2^{E-p}\epsilon(\alpha - h)$$

Because a result usually does not only consist of one floating-point operation different rounding errors need to be summed up. Assuming E and ϵ are independent of $\alpha - h$ leads to the following representation where R is the rounded and r the mathematical exact result and n the number of operations issued to obtain R. α_i is the random variable for the i-th operation. The function $g_i(d)$ describes the data and program used and is independent of α_i.

$$R = r + \sum_{i=1}^{n} g_i(d)2^{-p}(\alpha_i - h_i) + O(2^{-2p})$$

For unbiased rounding h is equal to zero and in the first order probabilistic model $O(2^{-2p})$ can be omitted leaving

$$R \approx r + \sum_{i=1}^{n} g_i(d)2^{-p}\alpha_i$$

as final description.

This leads to two basic hypotheses:

a) variables α_i are independently centred and uniformly distributed

b) first order model for R is legitimate

If both assumptions hold, all values obtained by DSA will be Gaussian distributed (see [Vig93]).

Basically this means that the four allowed rounding modes produces two different results with each being a valid representation of the exact result. Rounding up will give a slightly higher value and rounding down of course a slightly lower value. If only round down is used during a calculation the result will be underestimated. Same holds for rounding up, only that the result will be an overestimation of the exact result and it is obvious that the exact result lies in-between.

To get an estimation of the result rounding up and down will be done with a probability of 0.5 each (see [Che90]). The sequence of rounding up or down is randomly selected and therefore different for every execution. Therefore, N runs of a program will yield N results R_i with $i = 1...N$. N approximations are used to compose a mean value.

$$\overline{R} = \frac{\sum_{i=1}^{N} R_i}{N}$$

According to [Vig93], R_i is Gaussian distributed with mean value equal to the exact mathematical result, so with $i \to \infty$, $\overline{R}$ approaches to the exact result. Unfortunately such a high number of samples is not achievable. DSA makes it possible to estimate the number of accurate digits of a result for a certain confidence level by utilising Student's t-distribution known from statistics to compensate the lack of an extraordinary amount of samples. In the following formula τ_β is the value of Student's t-distribution (see A.1) with $N-1$ degrees of freedom and a confidence β that a sample will be less than τ_β.

$$C_{\overline{R}} = \log_{10} \frac{\sqrt{N} \cdot |\overline{R}|}{\sigma \cdot \tau_\beta}$$

σ is furthermore defined as

$$\sigma^2 = \frac{\sum_{i=1}^{N-1} (R_i - \overline{R})^2}{N-1}$$

If $C_{\overline{R}}$ is zero then $\overline{R}$ is to be considered insignificant. J Vignes introduced in [Vig93] the term *computational zero* to describe this state when a result has no informational content due to round-off error. If $C_{\overline{R}}$ is greater zero it represents the amount of accurate digits of the result for the given confidence level β. So if $C_{\overline{R}} = 2$ for $\overline{R} = 129.102$ then only the first two digits can be trusted and the last digits 9.102 may be contaminated by round-off errors.

2.3 CADNA Library

CADNA is a library that implements CESTAC, the predecessor of DSA, in ADA, C++ and Fortran. It is developed by Jean-Luc Lamotte, Fabienne Jézéquel, Laurent-Stéphane Didier and Jean-Marie Chesneaux. This library supports the development as well as the debugging of numerical applications ([CT]). The main features are

- estimation of round-off error
- detection of numerical instabilities
- checking of the sequencing of the program
- accuracy estimation of intermediate results

For this thesis only the C++ library is of interest. It provides two classes `double_st` and `float_st`. The postfix `st` stands for *stochastic type* and will be used as synonym for both classes. The stochastic types are meant to replace their C++ counterparts `double` and `float`. The mathematical $(+, -, /, *, ...)$ and the relational operators $(==, =>, <=, ...)$ are overloaded so that both classes can be used as if they were the build-in types. Furthermore, instances of these classes can be casted to `float`, `double`, `long`, `unsigned`, `int` and `short`. Most mathematical functions like `floor`, `ceil`, `pow` have been overloaded as well. One instance of a stochastic type contains three variables which represent the current state of this instance. All operators are applied to the three variables but with different rounding modes. Additionally, numerical analysis is applied to find numerical instabilities. The accurate result is represented by the mean value of the three variables of an instance. It is important that, as soon as one variable is using a stochastic type, at least the left side of an operator is a stochastic type too. This ensures that the accuracy information stored inside the variable is not lost. Floating-point operations form a virtual chain which is not to be broken. Table 2.2 provides an example.

Line	C++ source code	Stochastic type `a`	Stochastic type `b`	C++ type
1	`double d=0.0;`			d=0.0
2	`double_st a=2.0;`	a.x=2.0 a.y=2.0 a.z=2.0		d=0.0
3	`double_st b=3.0;`	a.x=2.0 a.y=2.0 a.z=2.0	b.x=3.0 b.y=3.0 b.z=3.0	d=0.0
4	`a=a*b;`	a.x=6.0 a.y=6.1 a.z=5.9	b.x=3.0 b.y=3.0 b.z=3.0	d=0.0
5	`d=a;`	a.x=6.0 a.y=6.1 a.z=5.9	b.x=3.0 b.y=3.0 b.z=3.0	d=6.0
6	`b=d;`	a.x=6.0 a.y=6.1 a.z=5.9	b.x=6.0 b.y=6.0 b.z=6.0	d=6.0

Table 2.2: Example of a broken stochastic type chain

Line 2 and 3 define the stochastic variables `a` and `b`. After the operation in line 4 the stochastic variable a contains three different values due to the round-off error (exemplary). The mean value represents the exact result. The operation in line 5 breaks the chain because the exact result is given to variable d but the stochastic information is lost. Variable b can not reconstruct the information. This is to be avoided.

CADNA is provided as library to be integrated into other applications. However, CADNA has some prerequisites that need to be met in order to function properly. First of all the application has to support C++. C applications that can not be compiled with a C++ compiler can not use CADNA. Using CADNA will result in an about four times bigger memory footprint of the application. Applications already operating at their memory limit can not afford this. Since CADNA replaces a type with a class incompatibilities with interfaces and other program parts need to be resolved which might not always be possible. A common example is a closed source third party library: the stochastic type cannot be passed

to the library and needs to be converted back to a standard type. As mentioned before, this will break the chain and the accuracy information is lost. Applications already using parallelisation libraries like MPI will have a serious problem with stochastic types because they contain three values instead of one. Normally, variables can be passed to other running instances through MPI provided methods. This only works for built-in types. Usually, class instances need to be converted to byte stream by the sender and reconstructed by the receiver introducing a huge overhead. A last limitation to mention is the extra time needed by CADNA to compute additional analysis steps which severely slows down execution. Applications already exhausting the maximal acceptable execution time without CADNA applied will drastically exceed this limit once CADNA is integrated.

3 Implementing DSA on CPU

For all simulations and implementations the soft- and hardware system described in appendix A.2.6 has been used.

The main focus of this thesis is to assess the applicability of DSA to the molecular dynamics simulation package GROMACS. With CADNA (see 2.3) there already exists a ready to use C++ implementation of DSA. The problem is that CADNA does a lot more than just analyse accuracy. All extra checks imply an additional contribution to the performance penalty. For long running simulations this easily means that simulation time grows from days (without modifications) to months if an accuracy analysis method is applied. So reducing simulation time has first priority. This is why CADNA in its original form won't be used. Nevertheless the core idea behind the library, to estimate the accuracy with only three executions as samples for the accuracy analysis, will be adopted. To realise the software solution a way to independently and randomly round up and down for each atomic floating-point operation is needed. This rounding behaviour will be addressed as *random rounding* (*rr*).

GROMACS currently contains 1,393,734 LOC[1]. Like most simulation software it has grown over decades and hundreds of person years of work have been put into. Refactoring the source code to integrate new functionality which needs to be in place at every atomic operation is not feasible. This basically leaves three other options to implement random rounding:

1. implement DSA in hardware
2. find a non source code intrusive method
3. automatically replace necessary parts in the source code

Option one is possible (see [CMb] and [CMa]) but not subject of this thesis. Option two can be implemented using the interface to the floating point unit presented from section 2.1 and is detailed in section 3.1. Automatically replacing source code as proposed by option three will be discussed in section 3.2. This chapter concludes with a comparison of the two methods in section 3.3.

[1] **L**ines **O**f **C**ode

3.1 Automated Code Insertion

Automated **C**ode **I**nsertion (short ACI) inserts the needed code for switching the rounding mode after each arithmetic operation automatically on assembly level. Chapter 2.1 already introduced a way to control the rounding behaviour of the FPU. The function `int fsetround(int)` is of peculiar interest. The function takes the rounding mode as argument, clears the FPU pipeline of any queued operations and sets the requested rounding mode. As it is IEEE 754 compliant the following rounding modes are supported:

- `FE_DOWNWARD`
- `FE_TONEAREST`
- `FE_TOWARDZERO`
- `FE_UPWARD`

On assembly level a function call to `fsetround()` translates to `fldcw WORD`. This call simply loads a special control word into the floating point registers (for more information about the structure of the control word please refer to [Int11] chapter 8, figure 8-9 through 8-12). Assembly code fortunately has a much clearer structure then C or C++. One line in assembly stands for one operation. Also there are no function and variables names to worry about. These circumstances make possible what in the C/C++ world would mess up the source code: search and replace. When searching for example `fadd` in an assembly source file one only gets the places where an floating-point addition operation is issued. In the C/C++ world the search result can be anything from a variable name to a comment. Now, if one would like to change the rounding mode after each floating-point operation, one would need to insert a call to `fldcw` with the appropriate control word as argument. So the first solution to applying random rounding to GROMACS is to write a script which switches rounding mode after each floating-point operation on assembly level.

First one has to understand the compilation process of GROMACS. It relies on a tool called `make`[2] which is configured by files called `Makefile` residing in each folder. Once called on the root directory it walks the folder tree and calls the configured compiler with the necessary arguments. The easiest way to manipulate the compilation of GROMACS therefore is to dock in-between `make` and the compiler. It is possible to provide `make` with custom compilers via the `CC` and `CXX` environment variables. Instead of providing the path to a normal compiler `CC` and `CXX` are now pointing to the script which does the magic namely `gccrr.py`.

Figure 3.1 illustrates the first step done by the script. It extracts the command line parameters for the compiler and replaces all arguments demanding an object file as target with the option to generate assembly files. The compiler then pre processes the C or C++ source

[2] http://www.gnu.org/software/make/

files and translates them down to assembly language. The script inserts the switching of rounding mode after each floating-point operation. Afterwards the target requested from the original command is created. The second step of the script only comes into play if the

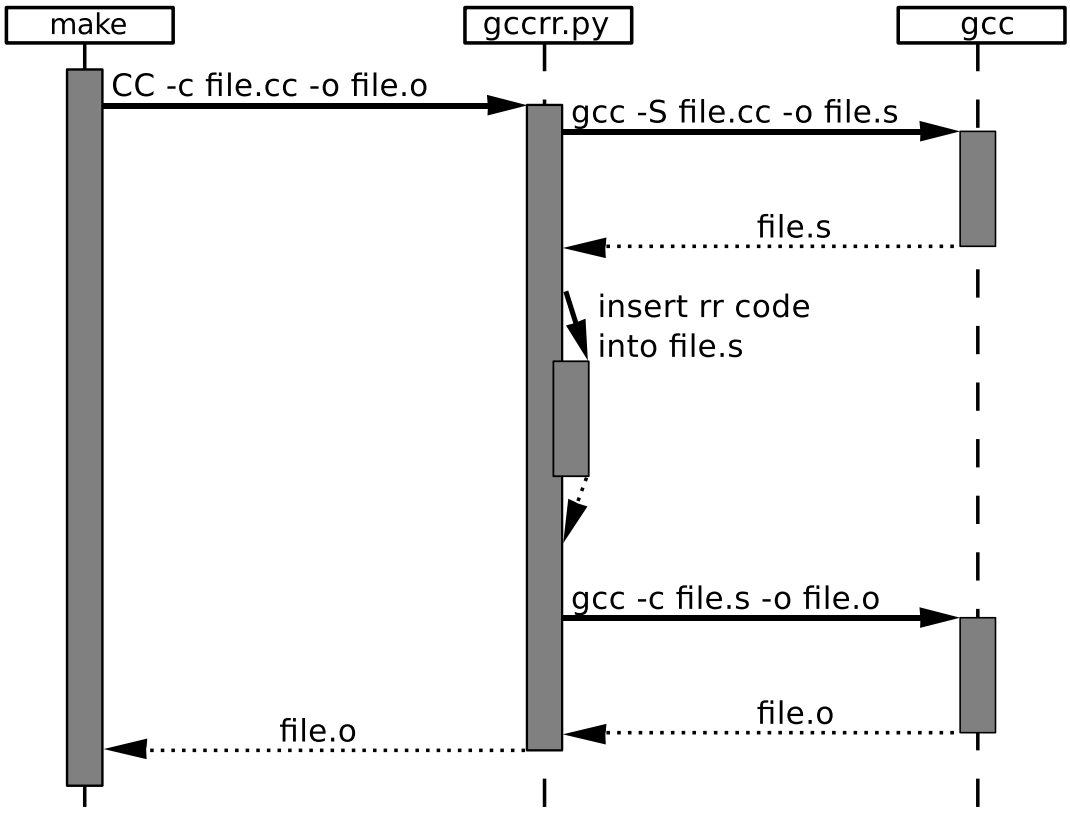

Figure 3.1: gccrr.py - Object file compilation

compiler needs to link several object or library files together. In this case a control word file is additionally linked to the target (see figure 3.2). Linking has to be done eventually to create a valid executable thus all executables will finally contain a control word object. It provides the microprocessor specific control words to set the rounding mode an some logic to randomly decide between round up and down.

Although a decision about rounding up or down could be done on the fly it is better from a performance point of view to generate a random value table at the beginning of the execution. If an execution reaches the end of a table it can be re-used from the beginning, guaranteeing a constant slow down independent of simulation length or number of floating-point operations. The initialization function also is provided by the control word file and a call to it can be inserted by script. The compiler script as well as the script to enable the initialization and further instructions can be found in the appendix at A.2.1 and A.2.4. A step-by-step

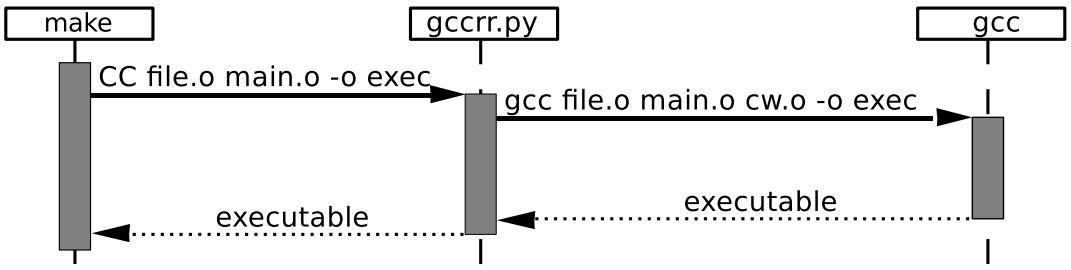

Figure 3.2: gccrr.py - Executable compilation

instructions on how to compile a random rounding binary of GROMACS can be found in appendix A.2.7.

A drop of bitterness is that the method seems not to be perfectly reliable in conjunction with GROMACS. In some rare cases the execution of a modified executable aborts with a segmentation fault due to access to uninitialized memory. Debugging revealed a questionable `if` condition comparing two floating-point values `if (r2 < rs2)` as the cause. If the expression evaluates to `true` a function is called which accesses an array with uninitialized values. Probably `r2` and `rs2` are correlated and their values are equal in a normal execution with round-to-nearest-even, and the expression evaluates to `false`. The array is not initialized because it is not needed in this case. With random rounding the correlation between the two variables is destroyed, and one of the variables has a slightly different value and the function is called under wrong prerequisites causing the segmentation fault. A solution to this problem is to extend the `if` condition like `r2 < rs2 && |r2-rs2| >` ϵ with ϵ set to a relatively low value. Unfortunately it is again not possible to do this for the whole code base of GROMACS within a short period of time.

Generally it would have been better to avoid such behaviour during development. A correct implementation should check data for initialization before usage.

3.2 Own Type Insertion

The second approach of implementing DSA in GROMACS is called **O**wn **T**ype **I**nsertion (short OTI) and uses an own type instead of the C/C++ types `float` or `double`. As this is exactly what CADNA does (see chapter2.3) it's C library is used to serve as basis. As mentioned before CADNA is very slow because of analyses which go beyond accuracy estimation. Therefore the overloaded operator function are stripped of their functionality. A downside of the random rounding implementation as presented in section 3.1 is that the FPU's pipeline gets emptied after each operation to change the rounding mode. To avoid

this a faster rounding technique is implemented. It is called NextFloat an presented in [CFD08].

At the beginning FPU rounding is set to round to zero. For positive values this equals rounding down and for negative values rounding up. To get rounding up for positive and rounding down for negative values one unit of least precision (ULP) needs to be added. This can be done by multiplying an operations result by $1.0 + 2^{-p}$ with p being the floating-point mantissa length. Instead of changing the rounding mode random rounding is done by multiplying a result with either 1.0 or $1.0 + 2^{-p}$. If a result of an operation would be exact, like $x = 1.0 * 2.0$, NextFloat will overestimate the result because of the additional multiplication.

Applied on a simple example calculating the dot product of two vectors in single precision, NextFloat rounding provides virtually a Gaussian distribution (see 3.3). Since the centre of the distribution should approach the mathematic correct result the mean value can be compared to a high precision result obtained with double precision. Table 3.1 shows, that the result from NextFloat rounding is much closer to the high precision reference sample than the result obtained with single precision.

Example	**Value**
Dot product high precision reference (double)	260,7141794713
Dot product NextFloat (mean over 4096 samples)	260,7141591236
Dot product (single)	260,7141418457

Table 3.1: Dot product precision comparison for OTI

Normally `a=a+b` would result in a `fadd` assembly operation. With the changes applied the code of listing 3.1 is execute instead. The comparison of operations shows that the rounding function is at least four times slower than the original code. Additionally there is the overhead introduced by converting C/C++ `float` to `float_st`, the function lookup in the VTable, pushing registers to the stack and so on.

```
1 x = a + b
2 x = x * {1.0 | 1.0+2^-23 } // example for single precision
3 index++
4 index = index & max_index
```

Listing 3.1: Rounding function for own type

The problem with `if` conditions as with Automated Code Insertion (section 3.1) can easily be avoided by applying the ϵ comparison to all overloaded relational operators.

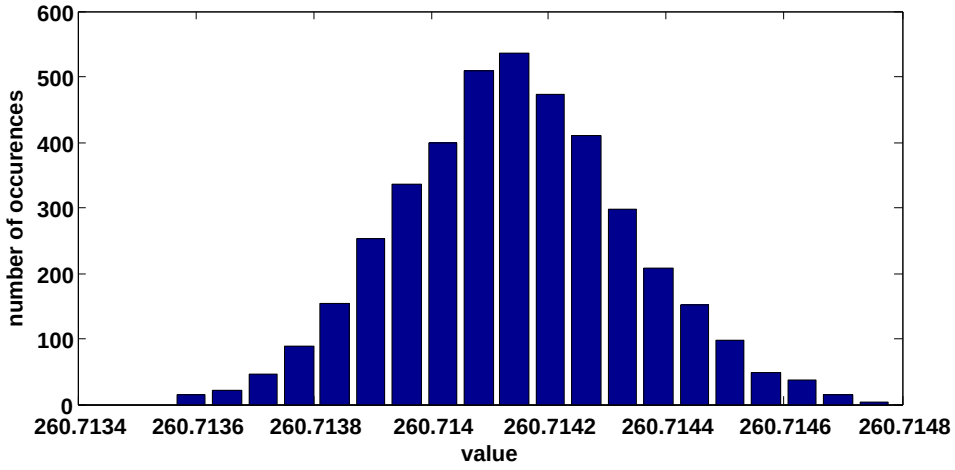

Figure 3.3: Value distribution for OTI dot product example

3.3 Method Comparison

Two methods have been introduced to implement DSA on CPU. Both have their area of application. Which one to use is a decision heavily depending on the use case. To simplify the decision-making this section provides a comparison of the different properties of the two methods. Table 3.2 provides a short summary.

Automated Code Insertion

Floating-Point Coverage Covers all floating-point operations regardless where they are hidden in the source code. Random rounding can only be deselected for sets of fp commands. If one addition operation should use random rounding, all addition operations in that source file will. If all operations should use random rounding this is the most thorough method.

Work Intensity Besides the testing after changing the source code which should be done for any method Automate Code Insertion is very easy to use. It only needs some changes to the build system and the new binary is ready to go. It is to note that the compilation process is slowed down by the random rounding compiler script. This is only of interest if the compiler script should be used in the development process.

Performance Because the pipeline has to be emptied before switching the rounding mode for each operation this method is really expensive in execution time.

Reliability In special cases ACI can cause problems during execution time. They can be resolved but this requires changes to the source code.

Development Since the code insertion happens at assembly level the inserted code can not be seen when debugging the C/C++ code. It is also very complicated to extend ACI. If there should be done more than just changing the rounding mode after each operation one has to be aware that the code is inserted without any knowledge of the state the program is currently in. For example might registers already be in use. The code inserted has to work under any thinkable precondition which makes the development very complicated. Any further extension will slow down the execution additionally.

Own Type Insertion

Floating-Point Coverage Only the floating-point operations involving a stochastic type will make use of random rounding. This makes it possible to control the usage of random rounding on a very fine level. If it is necessary to cover all operations replacing all relevant occurrence is either very unlikely or comes close to completely refactoring the package.

Work Intensity Replacing all floating-point types with a stochastic type is very work intensive. Fixing all resulting compiler errors is a complicated task. Especially if the source code of the software package is unknown to the developer. The corrections must not change the behaviour of the program.

Performance Compared to Automate Code Insertion, Own Type Insertion is really fast because it can make full use of the FPU pipeline.

Reliability No issues regarding the reliability where discovered during the analysis of this method.

Development Code can be added and debugged without limitations.

Short Summary

All properties in a short overview.

Automated Code Insertion	Own Type Insertion
Advantages	
• covers all fp operations • nearly no code changes • fast to apply	• safe solution • small performance penalty • extendible • debugable
Disadvantages	
• huge performance penalty • not perfectly reliable	• very work intensive • does not cover all fp operations

Table 3.2: Random rounding method comparison on CPU

4 Application of DSA in Molecular Dynamics Simulations on CPU

This chapter gives a brief introduction to GROMACS[1] and molecular dynamics simulations. GROMACS is a software package that solves the Newtonian equation of motion and is widely used for simulating biochemical molecules like proteins, lipids and nucleic acids. Utilising MPI[2], GROMACS can be executed on clustered environments.

First this chapter will give an introduction to molecular dynamics. Section 4.2 will present the example used in all simulations of this chapter. Then the results of the case study with two proposed DSA implementations will be discussed.

4.1 Fundamentals of Molecular Dynamics

Molecular dynamics (MD) is a field in computer simulation with the ambition to model the structure of molecules as well as the interaction of molecules among each other. Initial input data contains information about the type of molecules in the system, their position and their velocity, among others. Simulations are carried out in a step-by-step fashion. Each step calculates the forces acting on an atom

$$F_i = -\frac{\delta V}{\delta r_i}$$

by summing forces between non-bonded atom pairs

$$F_i = \sum_j F_{ij}$$

plus external forces and forces from bonded interactions and then solves newton's equation of motion

[1] http://www.gromacs.org/
[2] http://www.mpi-forum.org/

$$m_i \frac{\delta^2 r_i}{\delta t^2} = F_i, i = 1...N$$

to update the atoms positions before advancing on the time line [AAB+10]. The step size determines how much time passes between to steps. The smaller the step size is chosen, the closer the result will get to the natural worlds model.

4.1.1 Periodic Boundary Condition

Simulating takes place in a virtual box. Normally only a small number of molecules is put into the box (10.000 is still small) so that a majority of molecules are located near the surface of the box. To avoid unnatural behaviour in this region **P**eriodic **B**oundary **C**onditions (PBC) are applied [All04]. To simulate an infinite volume a replica of the box is put in every direction. If an atom exits the simulation box in the centre it will enter the very same box from the opposite direction keeping the total atom count constant.

4.1.2 System Properties

There are several key properties measured in a system:

- pressure
- temperature
- density
- potential energy
- kinetic energy
- volume
- box dimensions
- total energy

Density and **potential energy** are to highlight in this ensemble as they are later used in the case study.

Depending on the preferences of the simulation some properties are kept constant. The waterbox example from section 4.2 uses a barostat to keep system pressure constant by scaling the box vectors (see [Rüo8]). This implicitly has an influence on density which is determined by

$$\rho = \frac{m}{V}$$

with mass m and volume V of the system. Pressure again is calculated by

$$P = \frac{2}{V}(E_{kin} - \Xi)$$

with E_{kin} defined as

$$E_{kin} = \frac{1}{2}\sum_i^N m_i v_i \otimes v_i$$

v_i is the velocity of a particle and Ξ the virial defined as

$$\Xi = -\frac{1}{2}\sum_{i<j} r_{ij} \otimes F_{ij}$$

Consequently a systems density is indirectly linked to its kinetic energy.

Potential energy of a system is composed of various terms such as the Lennard-Jones and Coulomb interaction as well as bonded terms which are summed all together. The Lennard-Jones potential V_{LJ} is defined as

$$V_{LJ} = \frac{C_{ij}^{(12)}}{r_{ij}^{12}} - \frac{C_{ij}^{(6)}}{r_{ij}^{6}}$$

with $C_{ij}^{(6)}$ and $C_{ij}^{(12)}$ being atom pair dependent parameters. The Coulomb interaction V_c equals

$$V_c(r_{ij}) = f\frac{q_i q_j}{\epsilon_r r_{ij}}$$

As one can see from these parts the potential energy of a system is heavily depending on the particles position in the system.

4.1.3 Property Fluctuation

The fluctuation describes the difference between the mean and the current value.

$$\langle(\Delta x)^2\rangle^{\frac{1}{2}} = \langle[x - \langle x\rangle]^2\rangle^{\frac{1}{2}}$$

$\langle x\rangle$ denotes the average of x. The variance for a series N_x of values is given by

$$\sigma_x = \sum_{i=1}^{N_x} x_i^2 - \frac{1}{N_x}(\sum_{i=1}^{N_x} x_i)^2$$

4.2 Waterbox Example

Waterbox example will be the problem simulated in all cases where GROMACS is used. It models a rectangular box filled with water molecules and it behaves like a chaotic system where similar causes can end up in completely different result states (see [Avro8]). Periodic Boundary Conditions are applied as well as pressure and temperature coupling. The first period of time is needed to equilibrate the water mixture. This means the molecules need to find a position which could also be found in real life and therefore not violating any natural law. If this state is achieved, many properties of the model will reflect reality. With PyMol[3] it is possible to visualize the system state (figure 4.1). Oxygen atoms are coloured in red and

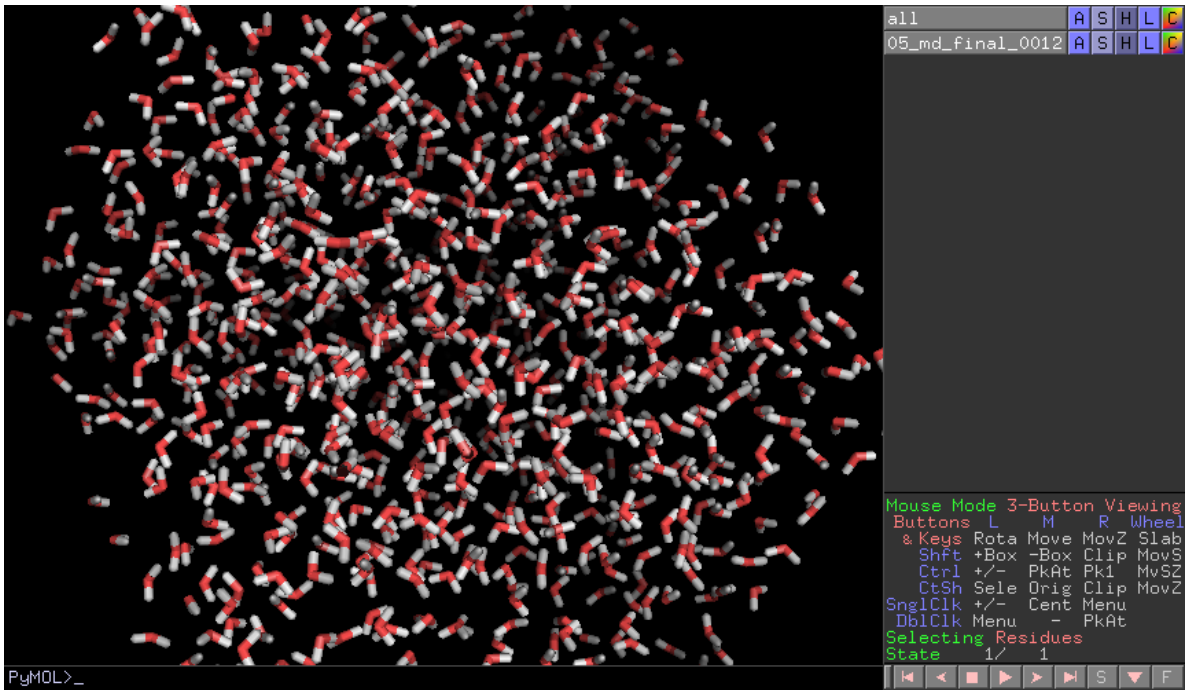

Figure 4.1: Waterbox visualized in PyMol

reside in the middle of the molecule with two adjacent hydrogen atoms on the left and right coloured in white.

[3] http://www.pymol.org/

4.3 Implementation Notes

Chapter 3 proposed two DSA implementation methods. Automated Code Insertion was chosen to be evaluated in a case study for section 4.4. A step-by-step guide of applying ACI to GROMACS version 4.0.7 is provided by appendix A.2.7. Own Type Insertion was done as a proof-of-concept and a working binary version of GROMACS was created. It only works with the latest major version of GROMACS (4.5.x) due to missing C++ support in prior releases.

SSE stands for **S**treaming **SI**MD **E**xtensions and was invented by Intel in 1993 [RCC+06]. **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) is a technique using data level parallelism and executes the same command on a block of data simultaneously thus boosting performance. GROMACS is able to use the SSE instruction set but the compiler script used with ACI is unable to change rounding for each atomic operation as they are all run in parallel. As a consequence GROMCAS needs to be compiled without this extension when applying ACI.

4.4 Waterbox Case Study with Automated Code Insertion

4.4.1 Execution Time

With ACI applied to GROMACS the implementation has to proof that it does what it was designed for. First test is whether the simulation still works. Comparing simulation data of the unadjusted version to the ACI version showed the same results. As expected the simulation time increased drastically. Figure 4.2 visualizes the drastic difference in execution

Version	**Optimization**	**Execution Times**			**Mean Time**
ACI single	unoptimized	933s	932s	934s	933s
ACI double	unoptimized	1.030s	1.026s	1.027s	$\approx$ 1.028s
single	unoptimized	60s	60s	61s	$\approx$ 60s
double	unoptimized	71s	71s	71s	71s
single	optimized	27s	27s	27s	27s
double	optimized	29s	29s	29s	29s
single	optimized + SSE	17s	18s	18s	$\approx$ 18s
double	optimized + SSE	18s	18s	18s	18s

Table 4.1: Execution times for 2.000 step waterbox

time. Interesting to note is that double precision seems to be as fast as single precision. This is a deception based on the short simulation length. When using longer simulations double precision will be 30% slower than single precision. The slow down factor of ACI compared

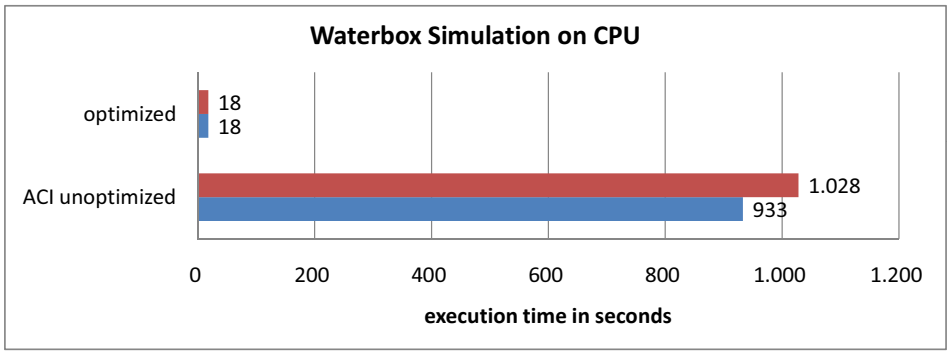

Figure 4.2: ACI runtime comparison

to the fastest, unadjusted version of GROMACS is $S_p = \frac{933s}{18s} \approx 52$ for one run. To perform an accurate digits estimation with DSA at least three independent results are necessary thus three simulations need to be executed. Normally one would start three simulations at the same time so that time to obtain three results won't grow. Three parallel simulations still need three times the resources a single simulation would occupy. Therefore, the total amount of resource consumption needs to be multiplied by three. Simulation time can also be considered as a resource which causes a combined growth in expenditure to apply ACI of 156x.

4.4.2 Effects of round-off error on a Chaotic System

Chaotic systems somehow contradict common believe in cause and effect. Lets take throwing a ball and measuring the distance as an example. Everybody will agree that two balls thrown with nearly equal force will hit ground in nearly the same area. Same consideration is not applicable to chaotic systems. With only nearly equal force thrown both balls can end up in totally different places. Yet thrown at exactly the same force they will sure land on top of each other. This also holds for the waterbox simulation. Two executions with exactly same parameters will end up in exactly the same result state. When random rounding comes into play no execution will exactly proceed as another resulting in probably totally different result states.

Figure 4.3 shows a waterbox after two steps of simulation. There are two simulations put on top of each other. The one obtained by an unchanged version of GROMACS is coloured green and the ACI version is coloured in red. Red can only be seen in place where the position of a molecule differs between the two simulations. For two simulation steps nearly

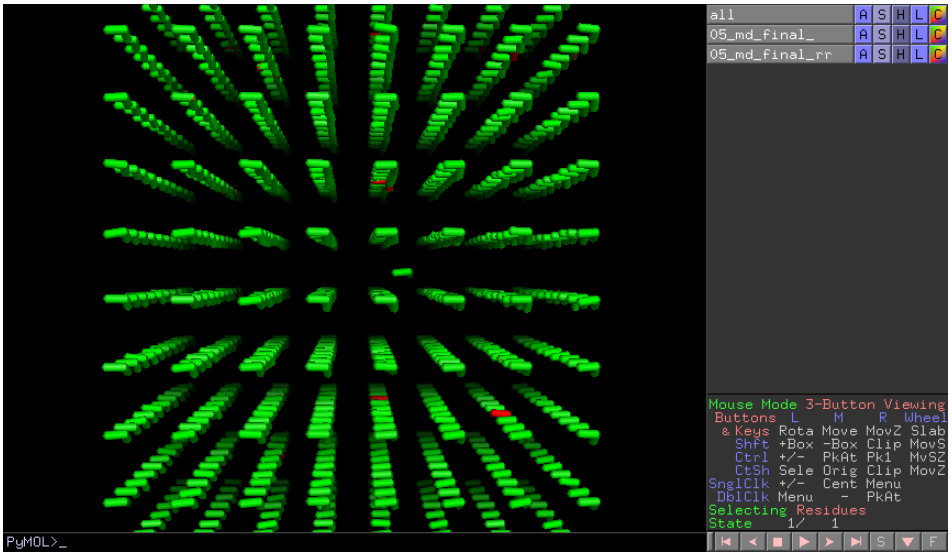

Figure 4.3: Waterbox after 2 steps

everything is green. After ten time steps (figure 4.4) first discrepancies can be observed.

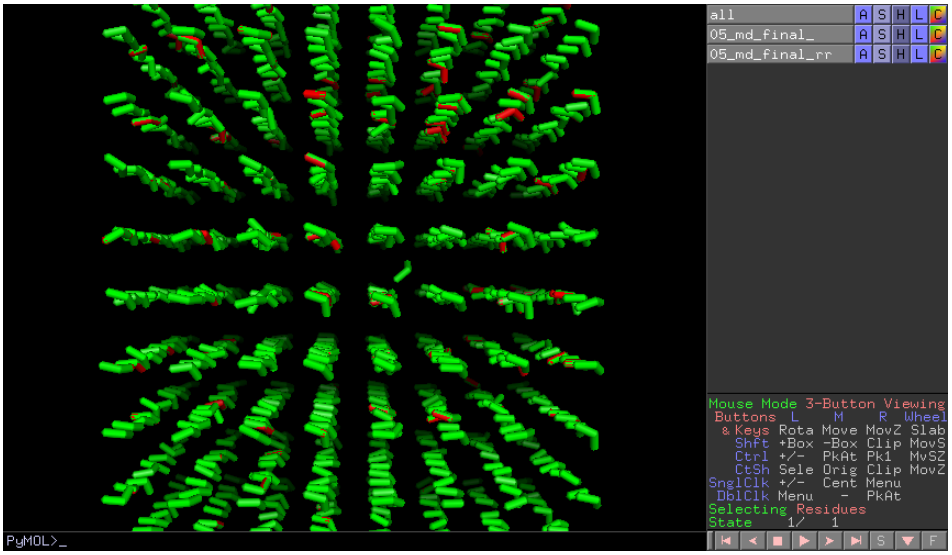

Figure 4.4: Waterbox after 10 steps

Figures for 40 and 200 time steps make clear that divergence will increase over time (figures 4.5 and 4.6). It is important to understand that both simulations represent a valid system since all physical and biomechanical laws are being obeyed.

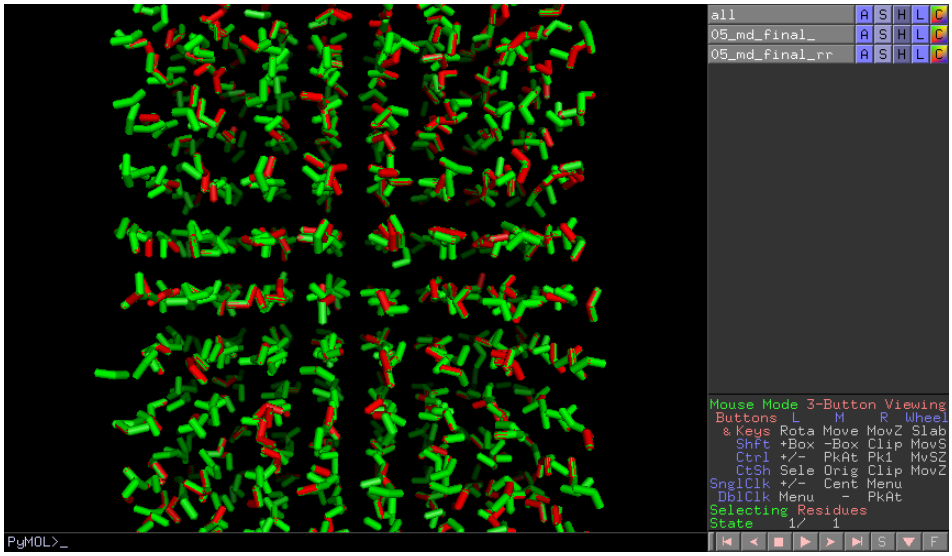

Figure 4.5: Waterbox after 40 steps

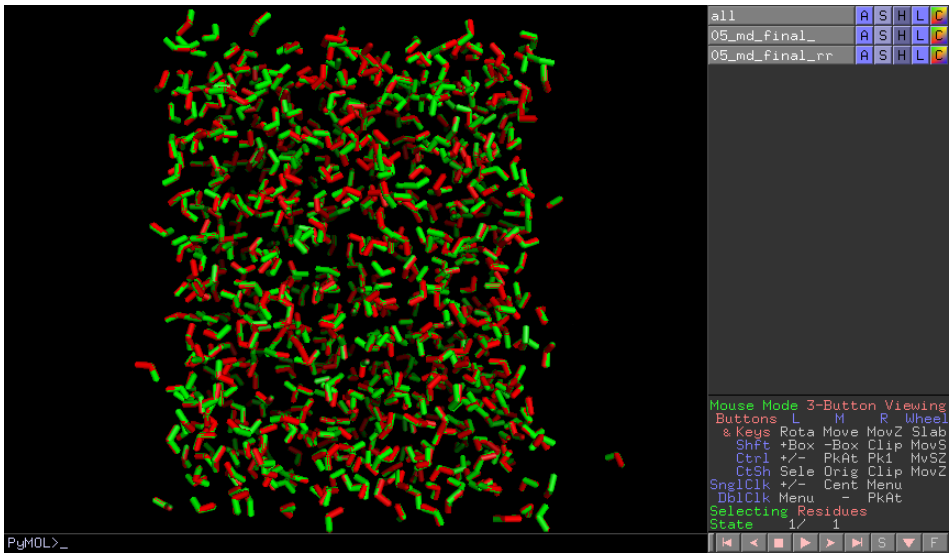

Figure 4.6: Waterbox after 200 steps

4.4.3 Accuracy Estimation of Density and Potential Energy

Estimating accurate digits was done for four different simulation set-ups. Simulation length was fixed to 80 pico seconds resulting in 20.000 time steps for a step size of 2 femto seconds. Second simulation series used a step size of 4 femto seconds. To reach a simulation length of

80 pico seconds the amount of time steps had to be cut to 10.000. For a step size of 6fs the amount of steps is further scaled down and reaches 5.000 for 8fs. Each series contains 16 simulation samples used to calculate the amount of accurate digits. Table 4.2 provides the exact values calculated with instructions provided by DSA. The amount of accurate digits

Step Size	**Potential Energy**		**Density**	
	single	double	single	double
2fs	2,671543225	2,736112016	2,286969324	2,339662333
4fs	2,678121365	2,843117985	2,527940688	2,224069343
6fs	2,756327267	2,805290527	2,279916378	2,391591876
8fs	2,77395379	2,704616572	2,099444145	2,165402125

Table 4.2: Accurate digits estimation for waterbox

ranges from about 2 to a little under 2,9. Potential energy is, according to these numbers, more accurate than density. Figure 4.7 plots accurate digits over step size. One can see that a change of step size nearly has no affect on accurate digits count. Plotting same parameters

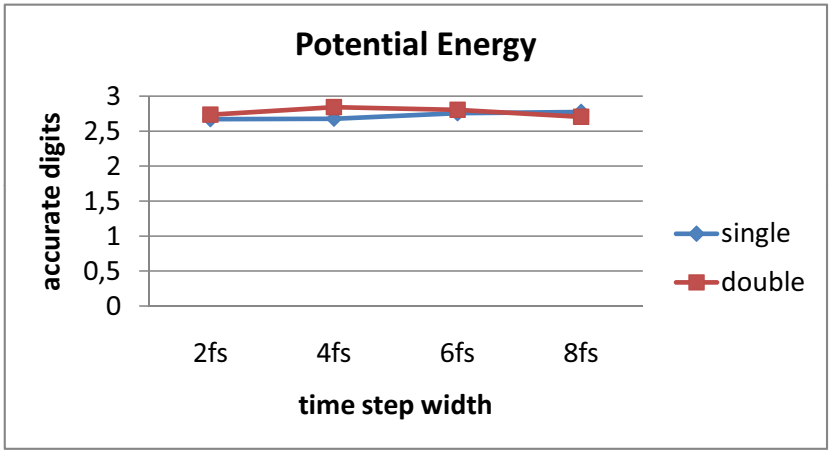

Figure 4.7: Accurate digits of potential energy

for density suggests that by increasing the step size accuracy decreases.

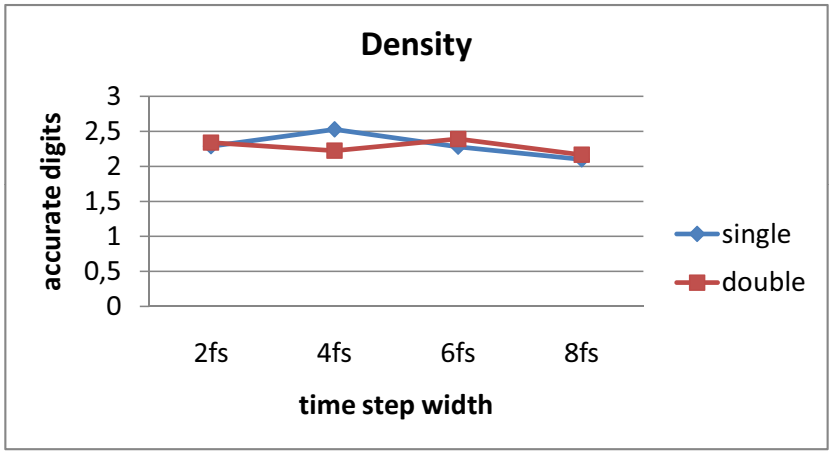

Figure 4.8: Accurate digits of density

4.4.4 Histogram

DSA assumes that the computed results are Gaussian distributed and centred at the mathematical correct result [Vig93]. If obtained results are still Gaussian distributed the method is believed to be compliant to DSA hypotheses.

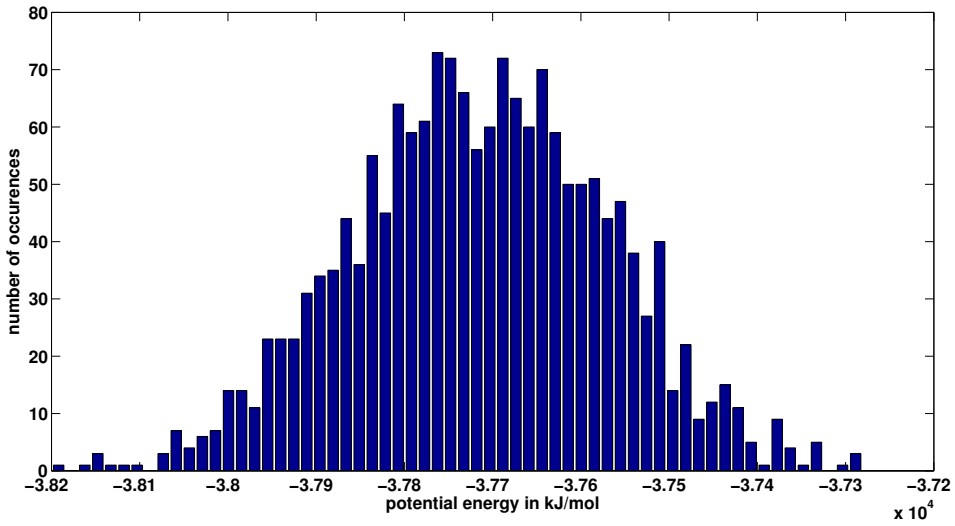

Figure 4.9: Result distribution for potential energy

To evaluate the distribution several thousand simulations were used to extract enough samples for a distribution analysis of the randomly rounded results. Figure 4.9 shows a histogram for potential energy. The x-axis values represent the calculated potential energy

for the system while the y-axis shows the number of occurrences. Same is plotted for density in figure 4.10. Lab experience has shown that even a rectangular shaped distribution

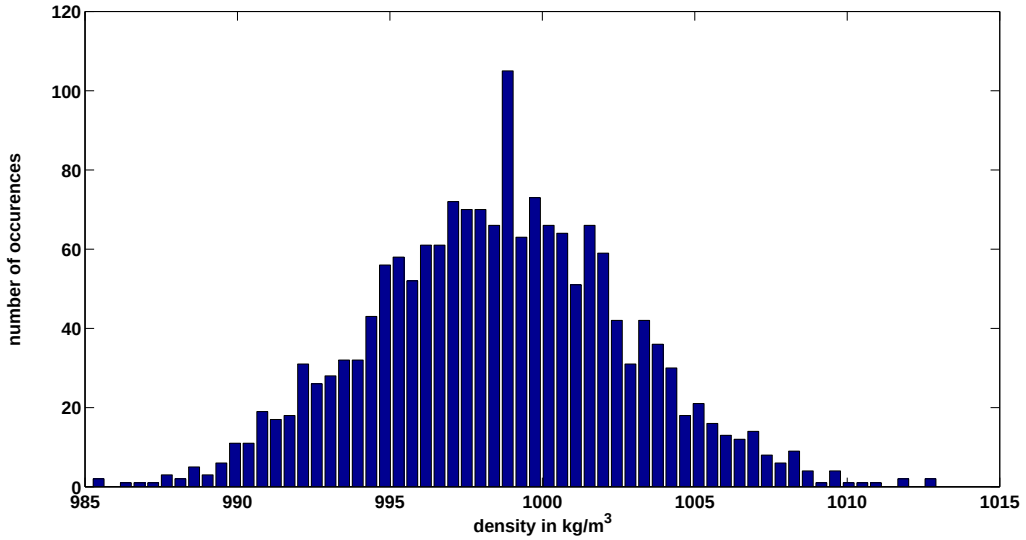

Figure 4.10: Result distribution for density

would provide acceptable results. Therefore both distributions show a good shape close to a Gaussian distribution.

4.4.5 Error Significance

Single precision data from the unchanged version of GROMACS could not be gathered for a time step width of 8fs. This is because the bigger the time step gets, the higher the possibility that a particle moves too close to another particle. Two particles can not be in the same place and to solve the situation a force is applied on each particle to accelerate them in opposite directions. Adding forces inflates the system with extra energy. If there is too much additional energy the simulation is aborted. For single precision this was the case at 8fs step width. Interestingly the ACI version of GROMACS simulated 8fs without problems.

A molecular dynamics simulation is not a static system and constantly in motion. The waterbox example uses pressure and temperature coupling resulting in varying box dimensions. Thus density and potential energy are not constant values either. Variance can be measured in a fluctuation calculation presented in section 4.1.3. Figure 4.11 plots the fluctuation for potential energy. Random rounding did not change the fluctuation of potential energy at all. For density the plot (figure 4.12) shows constant fluctuation for ACI enabled simulations. However, fluctuation for unchanged versions of GROMACS rises with increasing time step

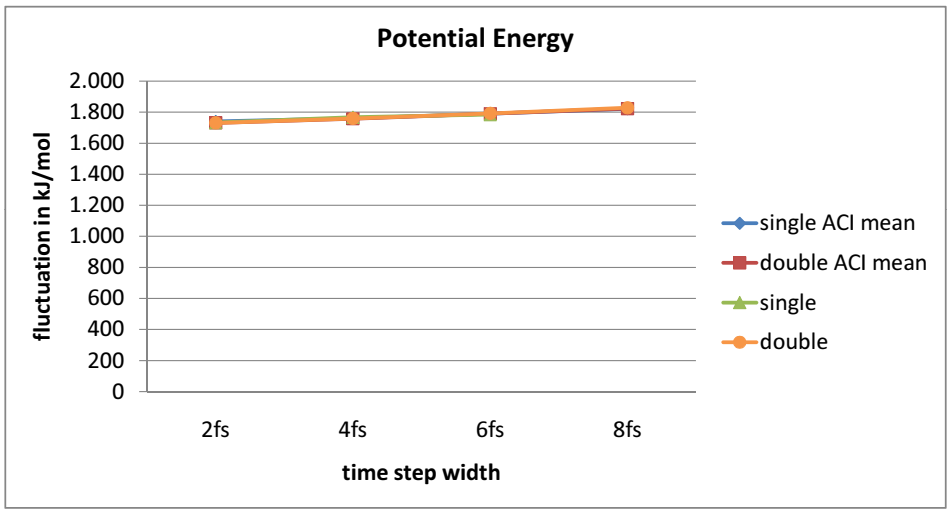

Figure 4.11: Fluctuation for potential energy

width which is a result of the same cause that makes it necessary to abort simulations described above.

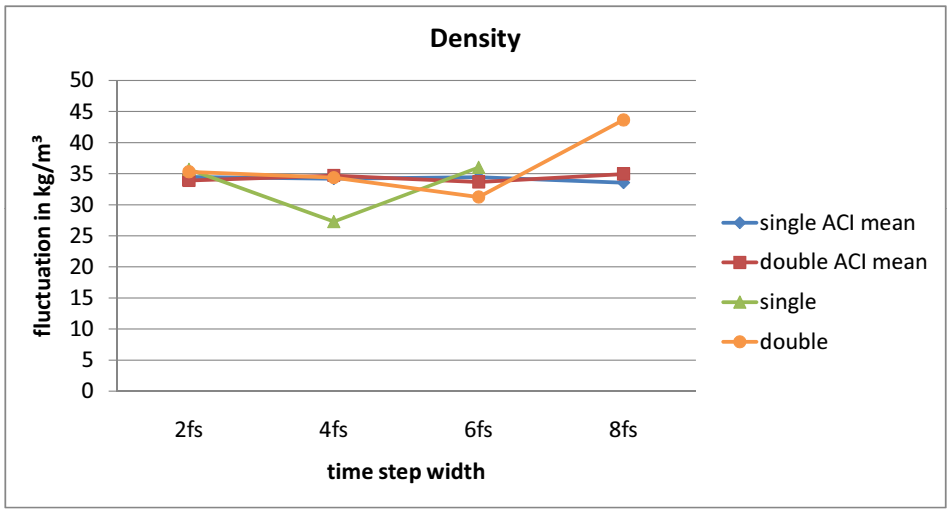

Figure 4.12: Fluctuation for density

The Institute of Technical Biochemistry at the University of Stuttgart is also using GROMACS to run molecular dynamics simulations. According to lab experiments a fluctuation of about 5% relative to the actual value of a measured property is acceptable. Figure 4.13 plots fluctuation of density divided by density over the four presented simulation set-ups. Values

obtained by GROMACS with Automated Code Insertion are averaged over 16 obtained samples. Clearly the application of random rounding has no effect on fluctuation of potential energy. Same information is plotted for density in figure

If 5% of fluctuation is acceptable this is a good limit for numerical accuracy as well. Only if numerical errors are bigger than fluctuation they pose a serious problem to MD simulations. Table 4.3 list relative numerical errors for given simulations. For both, potential energy

Step Size	**Potential Energy**		**Density**	
	Relative Numerical Error in Percent			
	single	double	single	double
2fs	0,213037852	0,183606471	0,516452848	0,457443717
4fs	0,209835341	0,143509951	0,296523633	0,596939967
6fs	0,175255934	0,156570332	0,52490852	0,405889787
8fs	0,168285311	0,197416491	0,795345548	0,683278688

Table 4.3: Relative numerical error in percent for potential energy and density

4.13 and density 4.14, the numerical error is under one percent. Relative numerical error in potential energy is stable for evaluated step widths. For density it seems to increase for bigger step widths.

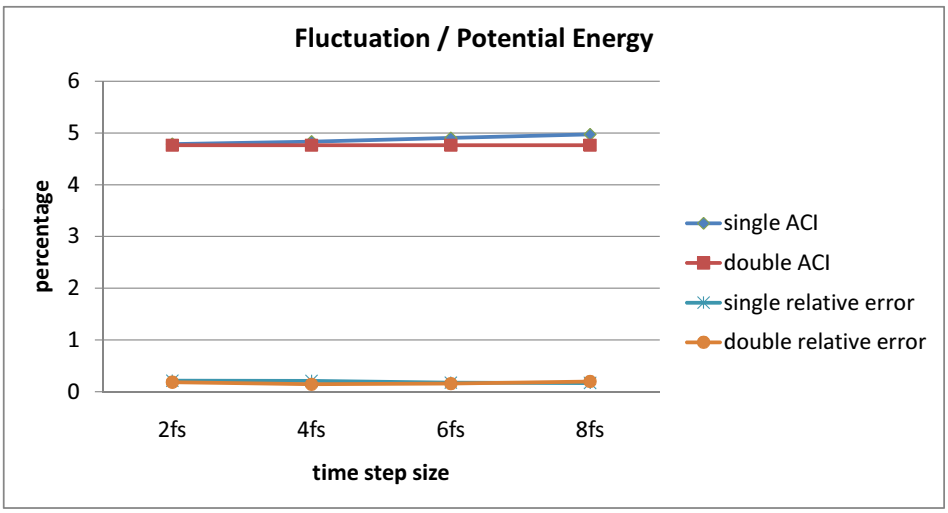

Figure 4.13: Percentage of simulation result for potential energy

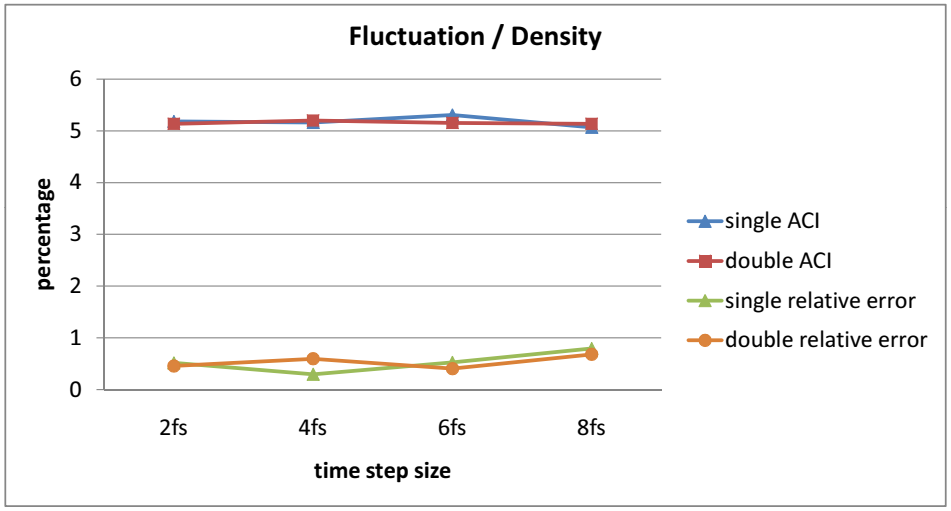

Figure 4.14: Percentage of simulation result for density

4.5 Applying Own Type Insertion to GROMACS

Applying own type insertion to GROMACS was a very work intensive task. Fortunately GROMACS can be compiled with either single or double precision. This means there already has to be a mechanism to switch a type to avoid duplicating source code. Listing 4.1 provides the relevant passage. This is very convenient as `real` only needs to be replaced by a stochastic type from OTI library.

```
1 #ifndef HAVE_REAL
2 typedef float        real;
3 #define HAVE_REAL
4 #endif
5 #ifndef HAVE_REAL
6 typedef double       real;
7 #define HAVE_REAL
8 #endif
```

Listing 4.1: simple.h - GROMACS type declaration

GROMACS's developers are proud their software is highly optimized. However, high optimization often means writing very specific code, using tricks and neglecting clean implementation. Therefore simply replacing a type by another will not suffice. Accordingly compiling GROMACS resulted in a lot of compiler errors and even more warnings. One problem was that not the whole package uses `real` as type. In some places **real** was only used to point to a specific implementation at linking stage. Listing 4.2 gives an example. Line 2 defines a function with `real` as argument. At linking time `real` is already replaced

with either `float` or `double` and the definition can be linked to according implementation. Replacing `real` with a stochastic type breaks this behaviour.

Also there were a lot of type casts involving `float` or `double` arrays. They only worked under the assumption that `real` is replace by a built-in type, whereas, after changing the type, line 3 and 6 won't work together.

```
1 //*** header file ***
2 void gmx_erf(real *); // original
3 void gmx_erf(float_st *); // after replacement
4
5 //*** source file ***
6 void gmx_erf(float *) { ... };
7 void gmx_erf(double *) { ... };
```

Listing 4.2: Problematic function declarations

Another problem arose from C++ integration code. Nearly all files contained a mechanism shown by listing 4.3. It properly defines C code passages if a C++ compiler is used. Both stochastic types, `float_st` and `double_st`, are C++ classes. If some inserted code is placed inside `simple.h` the example won't compile any more since C++ code is called from within a C code section. This is not hard to resolve once found but under millions of lines of code it can be hidden pretty damn good.

```
1  #ifdef __cplusplus
2  extern "C" {
3  #endif
4  #if 0
5  }
6  #endif
7  #include "types/simple.h"
8
9  ...
10
11 #ifdef __cplusplus
12 }
13 #endif
```

Listing 4.3: Dangling extern C

One last problem worth presenting is only a compiler warning (see listing 4.4). Still the program won't execute properly if the warning's cause is not removed.

```
1 source.c:17: warning: cannot pass objects of non-POD
2 through '...'; call will abort at runtime
```

Listing 4.4: Variable argument list warning

Functions with variable argument lists can not be called with classes which are not of POD type meaning they are not allowed to have any user defined functions. Functions like `printf` and `fprintf` used for outputting text use variable argument lists. Since GROMACS outputs a lot of information thousands of warnings needed to be fixed.

The list of problems is far from being complete. Provided examples should give an impression of the variety of issues that can occur when applying OTI to an existing software package. At first try without much knowledge about GROMACS source code it took about 18 to 20 working days to get a working version of GROMACS. As a proof-of-concept only replacing `real` surely was sufficient. For productive use a lot more work needs to be put into properly applying OTI. There are plenty program parts where `real` was not used thus no variable replaced by a stochastic type.

4.6 Lessons Learned

Waterbox example is no ordinary simulation due to its chaotic-system-like behaviour. Both Automated Code Insertion and Own Type Insertion were successfully applied to GROMACS. Based on ACI a case study was conducted. All properties observed showed no drastic changes when simulated with random rounding. Accuracy analysis of density and potential energy revealed a relative numerical error of under one percent which is significantly less than tolerated 5% of fluctuation.

Own Type Insertion was developed with aim of accelerating ACI. This goal was reached. Figure 4.15 illustrates the slowdown for applying ACI as well as for applying OTI. OTI is, as expected, more work intensive to implement.

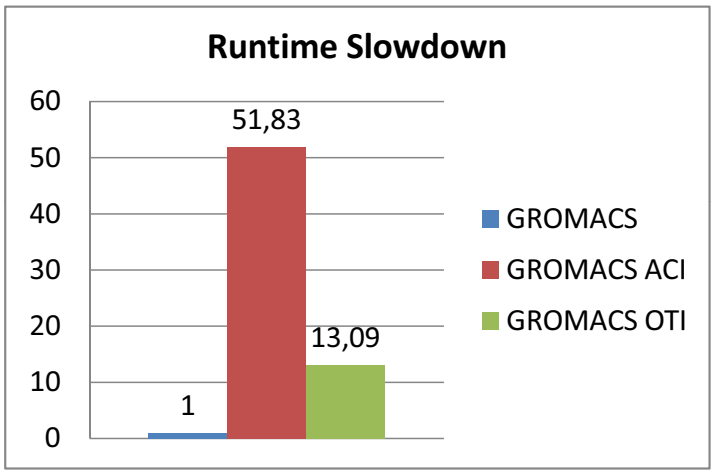

Figure 4.15: Performance penalty on CPU

5 Application of DSA in Molecular Dynamics Simulations on GPU

5.1 NVIDIA Graphics Cards

In 2010 NVIDIA published a new generation of graphics cards with the codename Fermi™. It is aimed to deliver high performance in computer games as well as in scientific computing. With full IEEE 754-2008 floating point support and eight times the double precision performance than its predecessor it tries to further broaden the use of graphics cards in the scientific sector. The following sections will clarify why Fermi™can accelerate computation intensive tasks by orders of magnitude.

5.1.1 Fermi Architecture

The first and by the time of writing fastest GPU based on Fermi™consists of 16 Streaming Multiprocessors (SM). They are arranged around the L2 Cache surrounded by the DRAM components (see figure 5.1). The Streaming Multiprocessor as shown in figure 5.2 contains

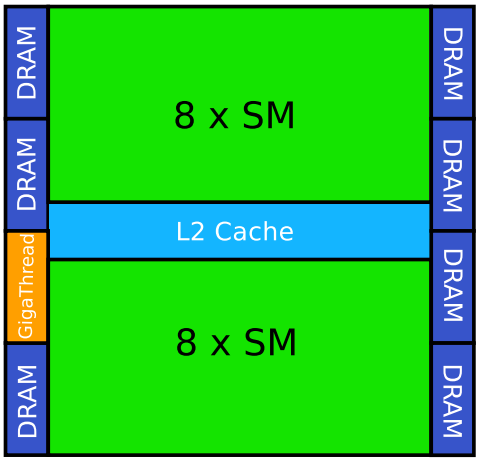

Figure 5.1: NVIDIA Fermi Architecture

32 CUDA cores. This sums up to a total amount of 512 CUDA cores per card. Each SM has

64 KB of L1 Cache, an Interconnect Network for inter-core communication a big register file and 16 load/store units (LD/ST) to read write data to and from the cache or DRAM. On the right-hand side the four Special Function Units are located (SFU). A SFU can handle one transcendental instructions as sin, cosine, reciprocal or square root per clock. A CUDA core

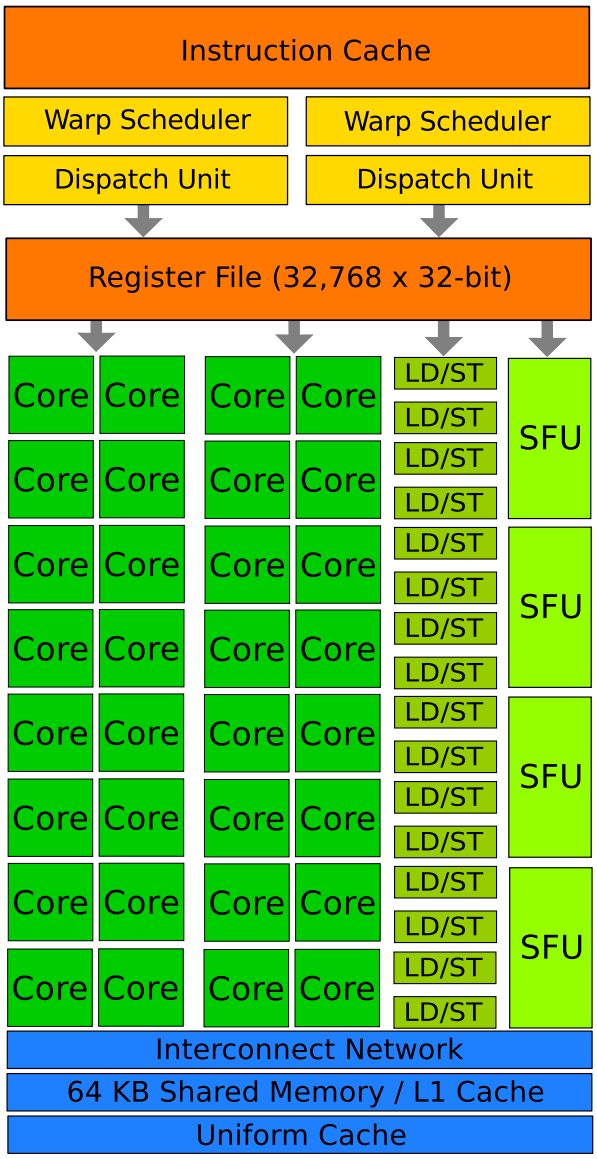

Figure 5.2: FERMI Streaming Multiprocessor

(figure 5.3) consists of a fully pipelined integer arithmetic unit (ALU) and floating point unit (FPU).

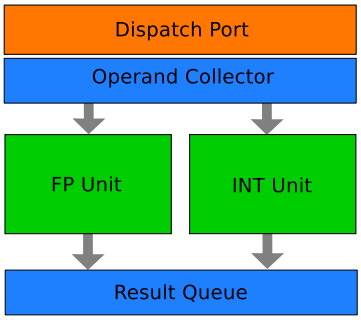

Figure 5.3: NVIDIA FERMI CUDA Core

5.1.2 Compute Unified Device Architecture

To accelerate an application with CUDA the computation intensive tasks need to be exported to so called kernels. A kernel is a function that is executed in parallel on the GPU. Figure 5.4 illustrates the organization of the threads. Each thread has its private memory. Threads are organized into blocks. Each block can resort to memory that is shared among all its threads allowing inter-thread communication. Blocks are furthermore organized into grids.

It is a developers task to choose the dimensions of grids and blocks. To fully utilize 512 available cores it is necessary to provide far more than one thread per core to hide memory latency (see [MSW95]). But one has to be cautious. If threads on one SM need more than the 32,768 provided registers one needs to reduce the amount of threads executed on an SM. Generally this is unlikely to happen. Unfortunately it is not always possible to dissect algorithms or functions into thousands of subtasks which are data independent. Naturally those tasks won't benefit from CUDA as much as highly parallelized ones.

NVIDIA provides a compiler front end called *nvcc*. It is used to compile CUDA source (file extension `.cu`) files into executables. It usually contains two types of source code.

host code Host code is written in C or C++. It handles the start up of the program and all interactions with components not residing on graphics cards (e.g. hard disk, RAM, keyboard, ...). It copies needed data to the graphics cards and calls a kernel to be executed on GPU.

device code A kernel is a good exemplary for device code. It is exclusively executed on the GPU. Device code can be linked to an executable either as PTX code, which is a kind of an assembly for GPU, or as binary GPU code called *cubin*. Both cases produce a global data array which is loaded to GPU by host code. It is also possible to create an object out of a *cubin* or PTX fragment which is loaded at runtime. For PTX code *just-in-time compilation* is used by the device driver to create a runnable binary version of the PTX file as soon as it is loaded onto the GPU.

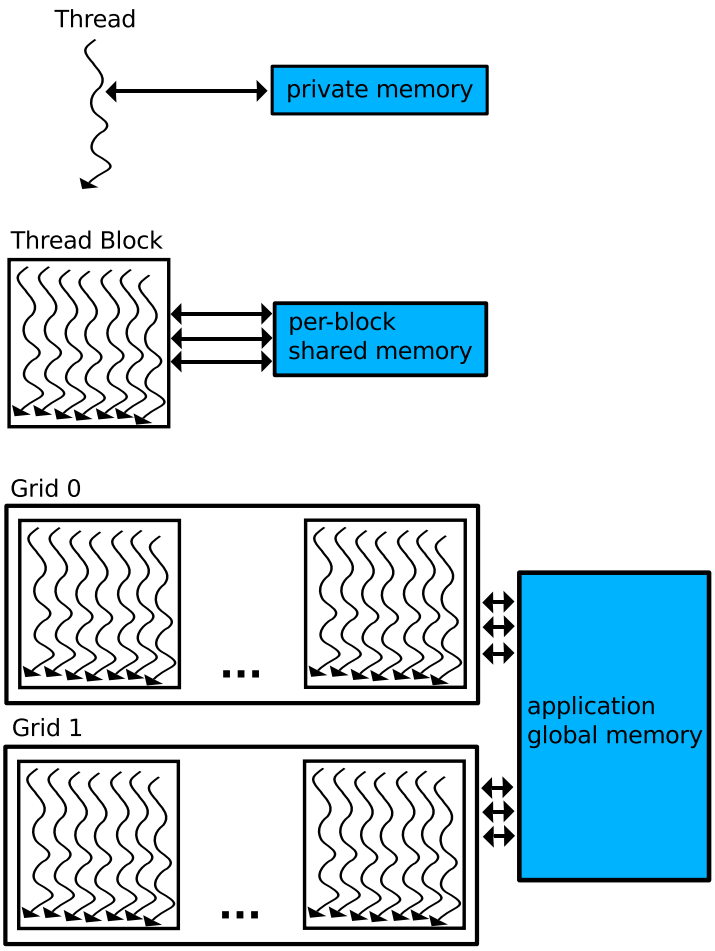

Figure 5.4: CUDA thread allocation

Figure 5.5 illustrates the structure of compilation stages used. First the pre-processor `cudafe` splits host and device code. Device code is further compiled by `nvopencc` to PTX code. `ptxas` finally creates binary device code which is placed in a *fatbin* format along a hash of the input files. Fatbin objects can be embedded into the executable or stored in file system to be loaded at execution time. Earlier extracted host code is compiled with a C/C++ compiler available on the system.

To distinguish between different available versions of CUDA enabled graphics cards NVIDIA introduced the compute capability model. It is defined by a major and minor revision number. Devices with same major revision number originate from the same GPU architecture. Minor revisions keep track of improvements inside a major line. It is possible to embed different compute capability enabled fatbins in an executable to leverage all features of the presented

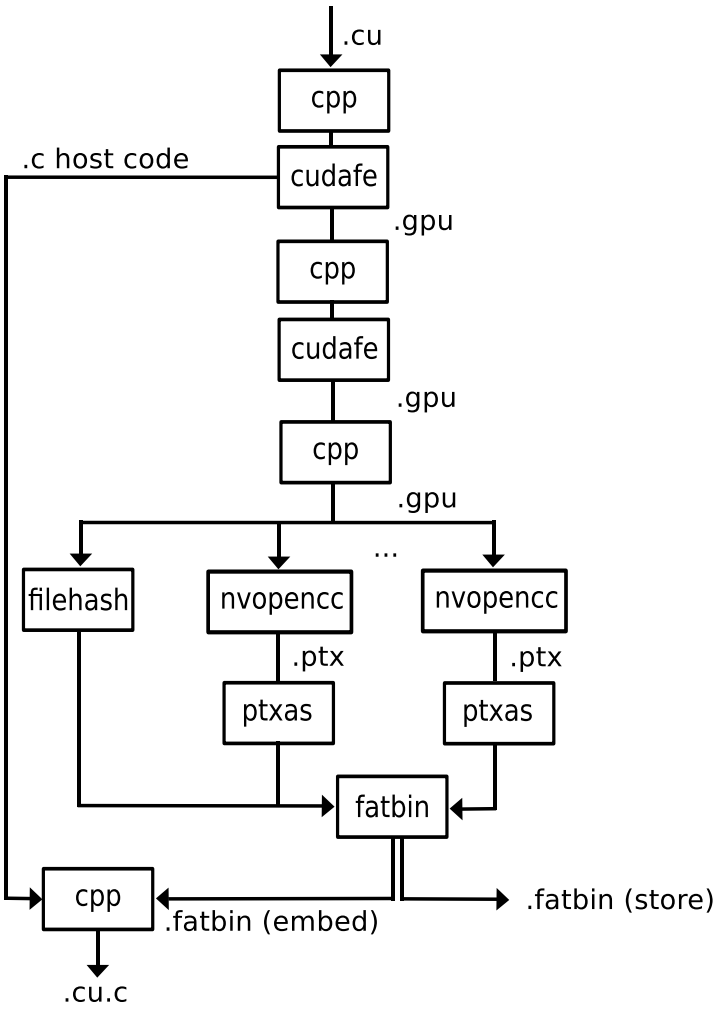

Figure 5.5: NVIDIA CUDA compiler stages

execution environment. A collection of features available per compute capability for current architectures can be found at [NVI10c].

5.2 Waterbox Performance on GPU

As mentioned earlier it is possible to achieve high acceleration in computations when moving from CPU to GPU. A simple matrix multiplication is a popular example. Depending on the matrix size a speedup of 300 can easily be achieved. Surely this is not possible for all applications.

Unfortunately the waterbox example uses the **P**article **M**esh **E**wald method (PME) during calculation of non-bonded potentials. Hereto a citation from GROMACS developer section:

"*Presently, the PME performance for a single GPU with ECC enabled roughly matches using GROMACS on all 8 cores of a cluster node. The primary bottleneck here is simply the reciprocal space grid algorithm which is highly accelerated on CPUs in GROMACS, while it is fundamentally harder to implement efficiently on GPUs - this is currently an area of intensive work. The GPU code actually uses 5th order interpolation internally, so you can usually improve performance a bit further by extending the grid/fourierspacing option.*"
(*Source: http://www.gromacs.org/Downloads/Installation_Instructions/Gromacs_on_GPUs*)

Thus the speedup by moving from CPU to GPU is nearly neglectable. There are other algorithms which highly profit from a GPU's computing power (see figure 5.6). The

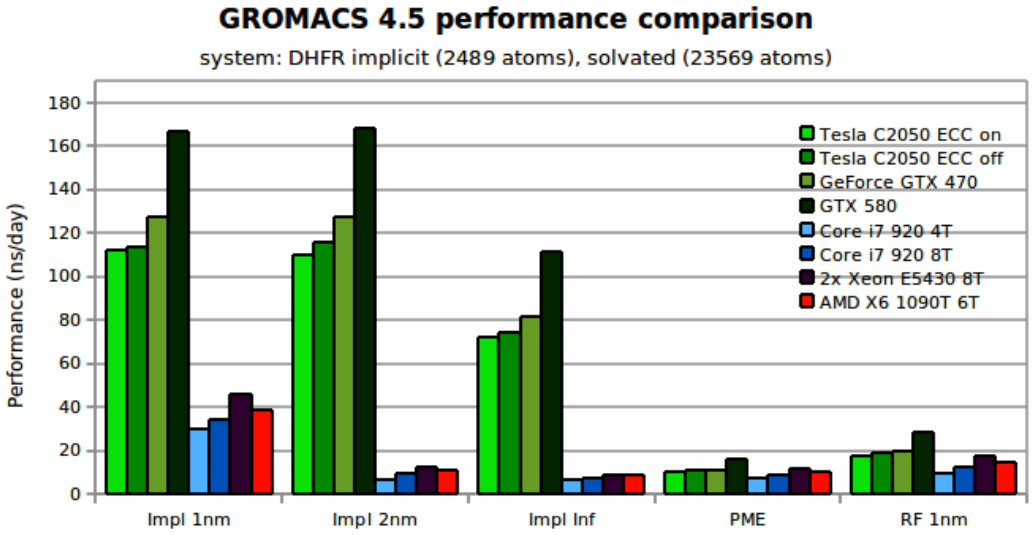

Figure 5.6: GROMACS GPU Benchmarks - *Source http://www.gromacs.org/gpu*

comparison already contains a sample from the GTX 580, NVIDIA's newest CUDA enabled graphics card, which yields a peak speedup of about 20. As expected, the waterbox example executed on GPU did not outrun the CPU version on the test system (A.2.6). Depending on conditioning of input data the GPU version did slightly better or slightly worse. Table 5.1 shows gathered values. There were four simulation set-ups executed starting with 2.400 molecules and doubling the molecule count for each of the remaining three simulations. Step size was set to 2 **f**emto **s**econds (fs) with 100.000 time steps resulting in a simulation time of 200 **p**ico **s**econds (ps). That a simulation with 9.600 atoms can be faster than one with 4.800 atoms is due to simulation overhead. There are similar effects on CPU for systems run on 64 CPUs and more. For 2.400 atoms CPU is faster than GPU. Here problem size is

Number of molecules	GPU	CPU
2.400	1.015 sec	774 sec
4.800	1.225 sec	1.202 sec
9.600	1.142 sec	1.353 sec
19.200	2.984 sec	2.812 sec

Table 5.1: Execution times for 100.000 step waterbox on GPU

too small to benefit from a highly parallel execution environment. Loading data to GPU takes more time than computing an actual result. In this case CPU will always be faster because no transfer overhead is present.

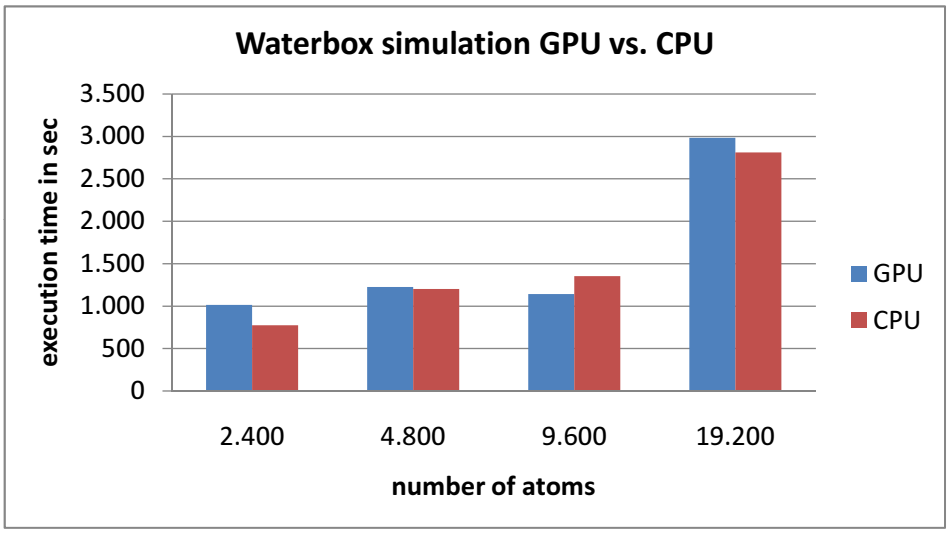

Figure 5.7: Waterbox runtime on GPU

5.3 Static Binary Rounding

Fermi^{TM}is the newest CUDA enabled architecture from NVIDIA and is closest to fully implementing IEEE 754-2008 yet [NVI10d], especially when single precision is concerned. In difference to CPU rounding it can not dynamically be changed during execution time. Rather there is a dedicated assembly command for each operation and rounding mode combination. For single precision floating-point multiplication those commands look like the following.

`mul.f32` Default to `mul.rn.f32`.

`mul.rn.f32` Multiplication with round to nearest even.

`mul.rz.f32` Multiplication with round to zero.

`mul.rm.f32` Multiplication with round to $-\infty$

`mul.rp.f32` Multiplication with round to $+\infty$

There are additional commands with combinations of `sat`, `ftz`, `approx` and `full` but since they do not have an influence on rounding they are not of interest.

No dynamic rounding means that DSA implementation one from CPU, Automatic Code Insertion, is not applicable here. Generally, a possibility to plug into the compilation process is present. Replacing NVIDIA's compiler by a script will have the same effect as placing a script between make and gcc for ACI. But with no command to control an operation's rounding mode there is nothing the script can insert.

Method two, Own Type Insertion, involved class operator overloading. CUDA cores are only fast for arithmetic operations and function calls or class operations do not count to their strengths. Because a drastic performance boost is expected by moving from CPU to GPU the OTI method won't be a good choice, although applicable. Still, CUDA provides rounded floating-point operators. Indeed they can't be chosen dynamically but during compilation time one is totally free to determine the kind of rounding applied to an operation. This means random rounding can be implemented analogous to ACI, only not during runtime but during compilation of the binary. For ACI each execution of a simulation has its own unique rounding sequence. Applied at compilation time each executable has its own unique rounding sequence which is static during execution time. Instead of running the same binary three times one needs to execute three different binaries each having a unique rounding sequence.

Table 5.2 gives exemplary binaries. All three still have the same program logic. They only differ in the command used for e.g. multiplication. This method will be referred to as **S**tatic **B**inary **R**ounding (SBR).

Operation	Executable A	Executable B	Executable C
a+b	round up	round down	round down
a*b	round down	round up	round down
a/b	round up	round up	round up
b+a	round down	round down	round up

Table 5.2: Example for three statically rounding binaries

Again, like with ACI, a script is handling the code changes making it as fast to implement as possible. Section 5.1.2 already approached CUDA's compilation stages and presented a dedicated step for processing CUDA assembly files in the PTX format. This makes it easier to apply changes to this step as only one tool with a sole purpose has to be replaced (see figure 5.8). The script can found in appendix A.3.1. Though the idea behind Static

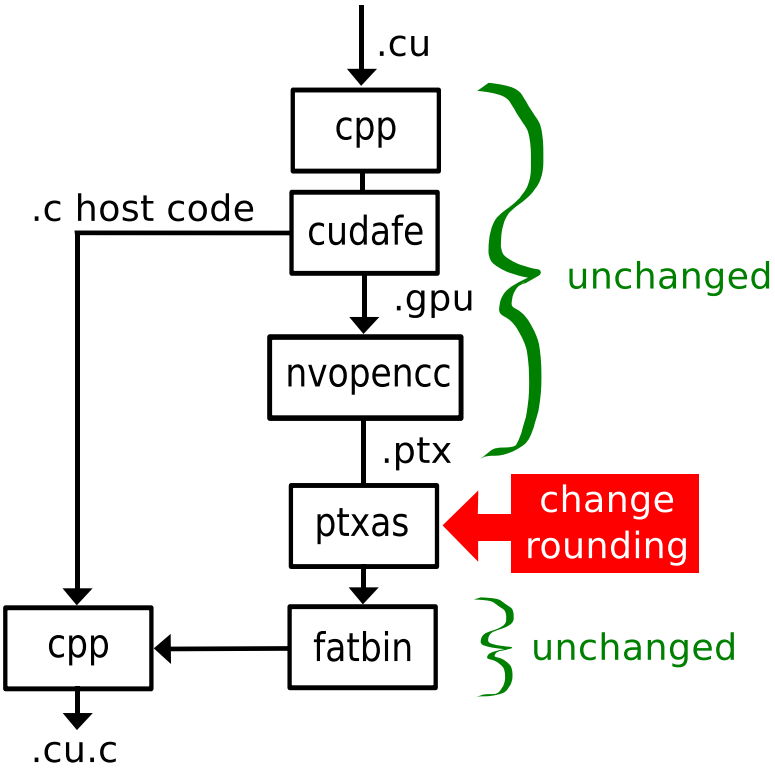

Figure 5.8: Changes in compilation stages for SBR

Binary Rounding is the same as behind Automated Code Insertion, still there is a significant difference. Rounding in a binary obtained by SBR is statically determined at compile time. When executed with CUDA a floating-point operation from binary file is (ideally) executed simultaneously on all CUDA cores. If this operation is determined to round up normally all threads of the same kernel will round up. A CUDA program which consists of only one floating-point operation embedded inside a loop has a rounding sequence length of one. SBR can only replace one operation with randomly chosen round up or down commands. The changed program only rounds up or down at all operations. Therefore the use of SBR is limited to programs with a decent amount of source code level floating-point operations.

GROMACS for GPU makes use of a library called OpenMM[1]. All molecular simulation related program logic which needs to run on GPU is provided by this library. Compiling said library and GROMACS with the SBR script worked fine. OpenMM provides several tests to verify the functionality of the installation. The adjusted version did pass all tests. Unfortunately GROMACS did not produce reasonable results for the waterbox simulation. Box dimensions are zero or negative, density calculated to infinite and potential energy only at a fraction of values observed on CPU. This behaviour was common to the changed and unchanged version of GROMACS. Therefore Static Binary Rounding can be excluded as perpetrator. Hints from the OpenMM development team point wrong data hand-over between GROMACS and OpenMM. This issue could not be resolved by end of this thesis.

To verify value distribution for SBR enabled binaries another example had to be used. A popular one for displaying CUDA's acceleration opportunities is matrix multiplication. In its original form matrix multiplication only uses two floating-point operations at source level: one addition and one multiplication. To inflate this example with more arithmetic operations loop unrolling was applied [HL].

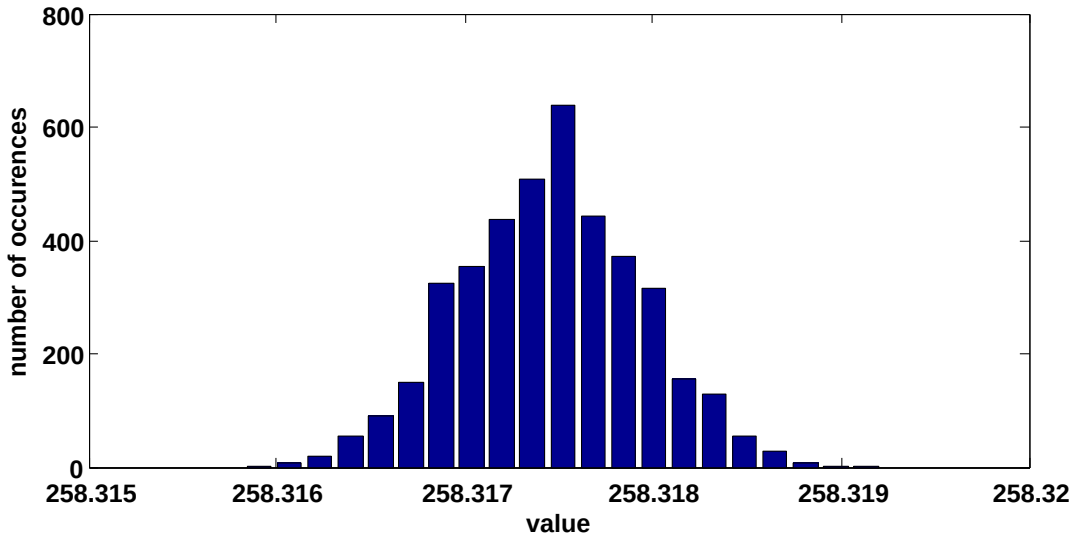

Figure 5.9: Value distribution for Static Binary Rounding

Afterwards each thread, and therefore each random rounding sequence, consisted of about 128 floating-point operations. Matrix size needed to be scaled as well to provide enough data for such excess of operations. Figure 5.9 plots the value distribution for a result component.

For a small example with only 128 occasions to generate a round-off error the resulting distribution is promising close to Gaussian distribution. For accuracy comparison the result

[1] https://simtk.org/home/openmm

of a dot product between a row and a column of the matrix is used. Again, double precision serves as high precision reference. The mean value of the results obtained with SBR is much closer to the high precision sample than the result computed with single precision (see table 5.3) making SBR a valid DSA implementation on GPU.

Example	**Value**
Dot product high precision reference (double)	258.3175263699
Dot product (mean over 4096 samples)	258,3174569085
Dot product (single)	258.3172912598

Table 5.3: Dot product precision comparison for Static Binary Rounding

6 Conclusion

This work assessed the applicability of numeric accuracy analysis based on DSA for CPU and GPU platform. Three methods were developed, two for CPU and one for GPU, to implement DSA in an existing software package. A case study was elaborated to review the effects of DSA on molecular dynamics simulation package GROMACS.

The challenge of implementing DSA in software consists in the fact that simulation tools, or computational software in a broader sense, grew over many years. Those applications are highly optimized and serve their purpose well. It is unlikely to convince users of working with or developers of implementing a new system only because an accuracy analysis should be integrated. So there is no chance of implementing such methods from the start. Best solution would probably be provided by hardware because a floating-point unit with random rounding capability is completely transparent to software running on top. Currently none of the big players in microprocessor industry offers products with this feature. Even if said products would exist it still were a question of monetary resources to replace a High Performance Cluster like JUGENE[1] by a pendant with random rounding capability. Thus, there are little alternatives to change existing software.

If changes need to be made it is desirable that these changes are made automatically. It allows to obtain results fast without investing too much resources beforehand. This might help to convince people of putting their software to a preliminary accuracy check. It also helps to use always the latest software version as new releases can be adopted with DSA fast. Last but not least an automatism makes less mistakes. Both Automated Code Insertion and Static Binary Rounding do provide this automation property.

In contrast, Own Type Insertion is not automatically applicable. There are steps like replacing a built-in type by a stochastic type which can be done by script or with refactoring tools of a development environment. Correcting broken code lines to reach a successful compilation still is a task to be done by hand. Unfortunately this has to be repeated for every new software version released, which is ultimately a question of costs. A benefit is the fast software rounding mode NextFloat which drastically accelerates Own Type Insertion in contrast to Automated Code Insertion.

[1] http://www.fz-juelich.de/portal/forschung/highlights/supercomputer

Biggest disadvantage of Static Binary Rounding is the limitation to programs with many floating-point operations on source code level. Besides that, it is the best solution from a technical point of view. It can be applied fully automatically, does not slow down the simulation and covers all floating-point operations.

An interesting question was whether the waterbox example would show a significant numerical error. Simulation results obtained with a GROMACS version using Automated Code Insertion suggest that the relative numerical error is under 1%. Fluctuation dominates changes of system properties with an amount of 5% of the actual value. More important is the fact that the implemented method did not break the simulation software and provided results usable for accuracy analysis.

Each of the three implementations has its own strengths and weaknesses, but none of them is suited for solving all problems at the same time. A decision can only be based on information about the intended use case one. Still, Automated Code Insertion and Static Binary Rounding are fast to apply and cover all floating-point operations. Static Binary Rounding combines these advantages with high execution speed and safest changes in code, making it the cleanest solution to implementing DSA from a technical point of view.

Table 6.1 summarizes properties of all three implementation methods.

CPU		GPU
Automated Code Insertion	Own Type Insertion	Static Binary Rounding
Advantages		
• covers all fp operations • nearly no code changes • fast to apply	• safe solution • small performance penalty • extensible • debugable	• covers all fp operations • absolutely no code changes • very fast to apply • no performance penalty • only assembly changes, no insertion
Disadvantages		
• huge performance penalty • not perfectly reliable	• very work intensive • does not cover all fp operations	• only works on big examples • only works on NVIDIA GPUs • needs compute capability ≥ 2.0

Table 6.1: Random rounding method comparison

6.1 Further Work

Currently Automated Code Insertion will not work on multi threaded programs. State variables and random rounding value table are globally defined allowing no simultaneous access. Using MPI is safe because each node has its own global data space. A common solution is to use mutual exclusion algorithms to avoid concurrent writes but will imply further slowdown.

On basis of Own Type Insertion alternative implementations for rounding floating-point operations can be assessed to find a robust and mathematic reasonable way to avoid emptying an FPU's pipeline after each operation. Round-by-multiplication has shown drastic performance gains compared to ACI and could serve as basis for such assessment.

NVIDIA's CUDA is under constant development. After adding limited support for C++ classes in device code with the last major release they promised to further complete C++ capabilities. It might become interesting to investigate porting Own Type Insertion to GPU to get random rounding on a fine grained level of floating-point operations.

A Appendix

A.1 Discrete Stochastic Arithmetic

A.1.1 Student's t-distribution

William Gosset introduced this statistical distribution in 1908. It is used when estimating a mean value of a normally distributed population with only few samples. The value is selected by the degrees of freedom (DoF) and the confidence interval.

DoF	75%	80%	85%	90%	95%	99%	99.5%	99.9%
1	1.000	1.376	1.963	3.078	6.314	31.82	63.66	318.3
2	0.816	1.061	1.386	1.886	2.920	6.965	9.925	22.33
3	0.765	0.978	1.250	1.638	2.353	4.541	5.841	10.21
4	0.741	0.941	1.190	1.533	2.132	3.747	4.604	7.173
5	0.727	0.920	1.156	1.476	2.015	3.365	4.032	5.893
6	0.718	0.906	1.134	1.440	1.943	3.143	3.707	5.208
7	0.711	0.896	1.119	1.415	1.895	2.998	3.499	4.785
8	0.706	0.889	1.108	1.397	1.860	2.896	3.355	4.501
9	0.703	0.883	1.100	1.383	1.833	2.821	3.250	4.297
10	0.700	0.879	1.093	1.372	1.812	2.764	3.169	4.144
11	0.697	0.876	1.088	1.363	1.796	2.718	3.106	4.025
12	0.695	0.873	1.083	1.356	1.782	2.681	3.055	3.930
13	0.694	0.870	1.079	1.350	1.771	2.650	3.012	3.852
14	0.692	0.868	1.076	1.345	1.761	2.624	2.977	3.787
15	0.691	0.866	1.074	1.341	1.753	2.602	2.947	3.733
16	0.690	0.865	1.071	1.337	1.746	2.583	2.921	3.686

Table A.1: Student's t-distribution values - *Source: http://www.stat.tamu.edu/~west/applets/tdemo.html*

A.2 GROMACS Related Works

A.2.1 Random rounding compiler script

The script used for automatically compiling GROMACS with random rounding arithmetic. When configuring the software package this script should be set as compiler in the CC environment variable. It then intercepts the normal compiler calls and reinterprets them to generate a binary with random rounding arithmetic.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Setting the GCCRR_DEBUG environment variable (to "true" or some other text) will make the script output additional information.

#----- IMPORT -----
import sys
from os import system
import os.path
import subprocess

#----- CONSTANTS -----
DEBUG=None                                          # Display debugging information.
FILE_CW_CONTROL_C = '/home/cmoetzing/Diplomarbeit/bin/cwc_64.c' # Path to the control word file.
COMPILER = 'gcc -m64'                               # Which compiler to use, also a good place to add additional compiler options.

CLEAN_ARGS = sys.argv[1:]                           # Exclude the name of the script from command line arguments.

FPU_COMMANDS_FOR_INJECT=[]

# This array contains all the floating point operations which will we bundled with a switch of the rounding mode.
# Please only input lower case commands. The script will also add every command as upper case.
FPU_COMMANDS = ["fadd","faddp","fsub","fsubp","fsubr","fsubrp","fiadd","fisub","fisubr"]
FPU_COMMANDS += ["fmul","fmulp","fdiv","fdivp","fdivr","fdivrp","fprem","fprem1","fimul","fidiv","fidivr"]
FPU_COMMANDS += ["fsqrt","fptan","fpatan","fsin","fcos","fsincos"]
FPU_COMMANDS += ["fexp","fyl2x","fyl2xp1"]
FPU_COMMANDS += ["fld","flds","fldl","fld1","fldl2t","fldl2e","fldpi","fldlg2","fldln2","fldz"]

# The source file extensions the script works with.
SOURCE_EXT = [".c",".cc",".cpp"]

```

```
32 # This is probably the most important part of this script. Those lines contain the assembly
       code to switch the rounding mode.
33 # 1) Save current content of of register.
34 # 2) Increment the array index.
35 # 3) Ensure the we do not run over the end of the array.
36 # 4) Load the index value to the register.
37 # 5) Get the new rounding word from the array.
38 # 6) Chop it to the right length.
39 # 7) Load the control word to the FPU register
40 # 8) Load the previous value back to the register to continue execution.
41 ROUNDING_CODE_LINES = """
42 \txchgq\t_RAX_(%rip), %rax
43 \tincq\t_RR_INDEX_(%rip)
44 \tandq\t$0xFFFFFF, _RR_INDEX_(%rip)
45 \tmovq\t_RR_INDEX_(%rip), %rax
46 \tmovzwl\t_RR_ARRAY_(%rax,%rax), %eax
47 \tmovw\t%ax, _RR_WORD_(%rip)
48 \tfldcw\t_RR_WORD_(%rip)
49 \txchgq\t%rax, _RAX_(%rip)
50 """[1:-1].split("\n")
51
52 INSTANCES=0    # For debugging purpose.
53
54 #----- FUNCTIONS -----
55
56 # Prints a status message.
57 def status(msg):
58         global DEBUG
59         if(DEBUG) :
60                 print "#>> %s" %(msg)
61
62 # Exits the script with an error if something bad happens.
63 def panic_code(msg, code):
64         global DEBUG
65         if DEBUG :
66                 print "########################################"
67                 print "#>> Panic: %s" %(msg)
68                 print "########################################"
69         sys.exit(code)
70
71 # Exits the script silently.
72 def panic(msg):
73         panic_code(msg, 1)
74
75 # Call the compiler with the given argunents. Arguments must be an array.
76 def call_compiler(args) :
77         cmd = "%s %s " % (COMPILER, " ".join(args))
78         # This resolves an issue with double quoted quotes.
79         if(cmd.find("\\\"") == -1) :
80                 cmd = cmd.replace("\"", "\\\"")
81         status(cmd)
```

```
82        # Open a subprocess to get the return code of the operation.
83        process = subprocess.Popen(cmd, shell=True, stdin=None, stdout=None, bufsize=0)
84        r = process.wait()
85        if r != 0:
86                panic_code(cmd, r)
87
88  # This function tries to find out whether gcc is called to compile something or just to get
      the version (gcc --version).
89  # If for example the autotools try to find out which verison of gcc is used this script
      should not interfere.
90  def has_compile_args(args):
91        ret = False
92        for arg in args:
93                if(arg == "-o"):
94                        ret = True
95                else:
96                        base,ext = os.path.splitext(arg)
97                        if(ext in SOURCE_EXT):
98                                ret = True
99        return ret
100
101 # If we create a executable out of some .o files we just need to call the normal command.
102 # The function first looks for an -o argument. If found, next is set to true meaning the
      next argument will decide.
103 # If the next argument contains a dot this means code injection will take place (no normal
      compile).
104 def do_normal_compile(args) :
105       if(not has_compile_args(args)):
106               status("No compile args found.")
107               return True
108       next = False
109       for arg in args :
110               if( next == True ) :
111                       if ( arg.find(".") == -1 ) :
112                               return True
113                       else :
114                               return False
115               if( arg == "-o") :
116                       next = True
117
118 # Compile the control word file to an object file. The object file is later linked to an
      executable containing some random rounding stuff.
119 def make_cw_control():
120       os.popen("%s -c %s%s.c -o %s.o" % (COMPILER,
              "",FILE_CW_CONTROL_C[:-2],FILE_CW_CONTROL_C[:-2]))
121       return FILE_CW_CONTROL_C[:-2] + ".o"
122
123 # Inserts the rounding code lines into the normal assembly code.
124 def translate_line(L,line):
125     # Append original assembly line to result.
126       L.append(line)
```

```
127         cmds = line.split(None,1)
128         # Ensure no empty lines are processed.
129         if ( len(cmds) != 0 and len(cmds[0]) != 0 ):
130                 # Remove AT&T syntax data size access markers from command.
131                 cmd = cmds[0]
132                 cmd1 = None
133                 if ( cmd[-1] in ["b","s","w","l","q","t"] ):
134                         cmd1 = cmd[:-1]
135         # Is cmd a floating point operation...
136                 if ( cmd in FPU_COMMANDS_FOR_INJECT or cmd1 in FPU_COMMANDS_FOR_INJECT ):
137                     # ... then append switching the rounding mode ...
138                         L += ROUNDING_CODE_LINES
139                         # ... and count how many switches were inserted (for debugging).
140                         global INSTANCES
141                         INSTANCES=INSTANCES+1
142
143 # Inserts the random rounding code into the assembly files.
144 def inject_rounding_changes(file_name):
145         file = open(file_name,"r")
146         text = file.read()
147         file.close()
148
149         old_lines = text.split("\n")
150         new_lines = []
151
152         for line in old_lines:
153                 translate_line(new_lines,line)
154
155         text = "\n".join(new_lines)
156         file = open(file_name,"w")
157         file.write(text)
158         file.close()
159
160 # Takes the requested compiler arugments and returns only those usable to generate an
        assembly file instead of an object file.
161 def extract_compile_args(args):
162         ret = []
163         # -S is the gcc option for compiling to assembly
164         ret.append("-S")
165         ommit_next = False
166         last_arg=""
167         for arg in args :
168                 keep = True
169
170                 if( ommit_next == True ) :
171                         ommit_next = False
172                         keep = False
173                 else :
174                     # -o is not of interest since we provide our own .s targets
175                         if( arg == "-o" ) :
176                                 ommit_next = True
```

```
177                             keep = False
178
179                     for s in [".o",".a",".so"] :
180                             # After a -MF or -MT we need to keep the targets.
181                             if ( arg.endswith(s) and not last_arg in ["-MF", "-MT"]) :
182                                     keep = False
183
184             # Linking will be done later.
185                     for s in ["-l","-L","-Wl"]:
186                             if ( arg.startswith(s) ):
187                                     keep = False
188
189                 if( keep ) :
190                         ret.append(arg)
191                 last_arg=arg
192         return ret
193
194 # Replace the source file endings with .s which we generated.
195 def extract_assemble_args(args):
196         ret = []
197         for arg in args :
198                 tmp = arg
199                 for ext in SOURCE_EXT :
200                         tmp = tmp.replace(ext, ".s")
201                 ret.append(tmp)
202         return ret
203
204 # If object files are linked together also link the control word file to it.
205 def extract_link_args(args):
206         ret=args
207         if(has_compile_args(args)):
208                 # compile c and/or add c file with static objects for our rounding magic
209                 cw_file = make_cw_control()
210                 ret.append(cw_file)
211         return ret
212
213 # Chain for compiling code with rounding mode switching.
214 def do_rr_compile(args, files, do_linking) :
215         # Make FPU code, no SSE.
216         args.append("-mfpmath=387")
217
218         # Compile withouth linking.
219         status("compile")
220         call_compiler(extract_compile_args(args))
221
222         # Inject rounding mode changes into .s files.
223         for (full_name,base) in files:
224                 inject_rounding_changes(base+".s")
225
226         # Assemble
227         status("assemble")
```

```
228         call_compiler(extract_assemble_args(args))
229 
230         # Link
231         if ( do_linking ):
232                 status("link")
233                 call_compiler(extract_link_args(args))
234 
235 # Initialization at beginning of script.
236 def init() :
237     # Add upper version of FPU commands.
238         for cmd in FPU_COMMANDS:
239                 FPU_COMMANDS_FOR_INJECT.append(cmd)
240                 FPU_COMMANDS_FOR_INJECT.append(cmd.upper())
241     # Handle debugging variable.
242         try:
243                 dbg = os.environ['GCCRR_DEBUG']
244         except KeyError:
245                 dbg="false"
246         if(len(dbg.strip()) != 0):
247                 global DEBUG
248                 if dbg == "true" :
249                         DEBUG = True
250                 else:
251                         DEBUG = False
252 
253 # Determine whether a link and a compile (like gcc main.c -o main.exe) takes place.
254 def link_and_compile_at_same_time(args):
255         if(do_normal_compile(args)) :
256                 for arg in args:
257                         base,ext = os.path.splitext(arg)
258                         if(ext in SOURCE_EXT):
259                                 return True
260         return False
261 
262 # Try to determine whether gcc is called from inside the configure script from autotools.
263 # This is necessary since the interference of this script does not work well with some tests
        done by autotools.
264 def is_configure(args):
265         for arg in args :
266                 if(arg.find("conftest") != -1) :
267                         return True
268         return False
269 
270 #----- MAIN -----
271 
272 init()
273 
274 if(is_configure(CLEAN_ARGS)):
275         call_compiler(CLEAN_ARGS)
276         sys.exit()
277 
```

```
278 if(link_and_compile_at_same_time(CLEAN_ARGS)):
279         panic("Cannot link and compile at the same time.")
280 
281 if ( do_normal_compile(CLEAN_ARGS) ) :
282         if(link_and_compile_at_same_time(CLEAN_ARGS)):
283                 panic("Cannot link and compile at the same time.")
284         status("Skip rr compile.")
285         call_compiler(extract_link_args(CLEAN_ARGS))
286 else :
287         status("Do rr compile")
288         do_linking = True
289         ommit_next = False
290         files = []
291 
292         for arg in CLEAN_ARGS :
293                 if ( arg in ["-c","-S"] ) :
294                         do_linking = False
295                 base,ext = os.path.splitext(arg)
296 
297                 if ( ext in SOURCE_EXT ) :
298                         files.append((arg,base))
299 
300         do_rr_compile(CLEAN_ARGS, files, do_linking)
301 
302 status("Injected rr-snippet %i times." %(INSTANCES))
```

Listing A.1: gccrr.py - Automated Code Insertion compiler script

A.2.2 Get FPU rounding control words

This file outputs the control words for the different rounding modes. The control words then can be put into a random rounding control file (A.3). To compile the source file execute

```
gcc get_rounding_words.c -o get_rounding_words -lm
```

The rounding modes are output in hexadecimal format:

```
Upward:  B7F
Downward:  77F
Nearest even:  37F
To zero:  F7F
```

Since the control words might vary on different hardware architectures it is important to verify using the right ones before compiling a software package with the gccrr.py script from A.1.

```c
#include <stdio.h>
#include <fenv.h>

long value = 2;

int main() {
    printf("Upward: ");
    fesetround(FE_UPWARD);
    __asm__("fstcw value");
    printf("%lX\n", value);

    printf("Downward: ");
    fesetround(FE_DOWNWARD);
    __asm__("fstcw value");
    printf("%lX\n", value);

    printf("Nearest even: ");
    fesetround(FE_TONEAREST);
    __asm__("fstcw value");
    printf("%lX\n", value);

    printf("To zero: ");
    fesetround(FE_TOWARDZERO);
    __asm__("fstcw value");
    printf("%lX\n", value);
}
```

Listing A.2: get_rounding_words.c

A.2.3 Random rounding control file

At the start of an execution of a program which was compiled with gccrr.py (A.1) the sequence of rounding control word needs to be generated. Done in this file it is automatically compiled and linked to the program. In this case the control words are valid for a x86_64 CPU. An important value is the size of the control word array. The bigger it is, the longer the initialization takes. If all numbers where used the counter is set to zero and a second iteration begins. If the size is changed make sure the line in the gccrr.py script (A.1) which resets the counter is adapted as well.

```c
#include <sys/time.h>
#include <stdio.h>

// Number of elements in the array. Remember to set andq in gccrr.py accordingly.
#define RR_ARRAY_LENGTH 0x1000000

// Temporary storage for registers during injected asm.
unsigned long _RAX_ = 0;
```

```
9
10 // Array to store a random sequence of round up and down codes.
11 unsigned short _RR_ARRAY_[RR_ARRAY_LENGTH];
12 // Index of current position in _RR_ARRAY_
13 unsigned long int _RR_INDEX_ = 1;
14
15 // Store the currently used rounding word.
16 unsigned short _RR_WORD_ = 0x0b7f;
17
18 // Random number generator.
19 static unsigned int lfsr = 0xFFFFFFFFu;
20 static unsigned int get_rnb()
21 {
22        // taps: 32 31 30 10; characteristic polynomial: x^32 + x^31 + x^30 + x^10 + 1
23        unsigned bit = ((lfsr >> 31) ^ (lfsr >> 30) ^ (lfsr >> 29) ^ (lfsr >> 9) ) & 1;
24        lfsr = (lfsr >> 1) | (bit << 31);
25        return bit;
26 }
27
28 // Called once at startup of application to fill the array equally with rounding up and down
       control words.
29 void enable_random_rounding()
30 {
31        fprintf(stderr, "Init rr array.\n");
32        struct timeval time;
33        unsigned i;
34
35        // The two used rounding control words.
36        unsigned short cws[] = {0xb7f,0x77f};
37
38        // This is the seed of the random number generator. If same control word distribution
              is needed twice, set this to a fixed value.
39        gettimeofday(&time,0);
40
41        lfsr = 1000*time.tv_sec + time.tv_usec;
42        for ( i = 0; i < RR_ARRAY_LENGTH; ++ i ) {
43               _RR_ARRAY_[i] = cws[get_rnb()&1];
44        }
45        fprintf(stderr, "RR enabled.\n");
46 }
```

Listing A.3: cwc_64.c - Random rounding control word file

A.2.4 Automatic Code Insertion enabler script

As mentioned in A.3 the array holding the control words needs to be initialized at a programs start. Therefore a function call needs to be inserted in the main function. This is done by this script automatically. To use it just point it to the root folder of the source code.

```
python enable_rounding.py /tmp/gromacs-4.0.7/
```

The script informs about the files where random rounding was enabled

```
Enabled rr in '/tmp/gromacs-4.0.7/src/kernel/gmxdump.c'.
```

and where the regular expressions might have missed a main function (which hopefully never occurs).

```
WARNING: grep my have missed a main function in '/tmp/gromacs-4.0.7/src/nm.c'.
```

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

#----- IMPORT -----
import sys
from os import system
import os.path
import subprocess
import re

#----- CONSTANTS -----

# Function to call from main.
RR_CALL="enable_random_rounding();"
# Definition of function which is linked to the file later.
RR_DEF="""extern void enable_random_rounding();
"""

# Exclued the name of the script from the command line arguments.
ARGS = sys.argv[1:]

# Regular expressions
RE_FILES=re.compile("^.*(\.c$|\.cpp$|\.cxx$|\.cc$)")
RE_MAIN=re.compile('[\s]*main[\s]*[\\(]{1}[\s]*int[\s]+argc.*\\)')
RE_MAIN_CHECK=re.compile('[\s]*main[\s]*\(')
RE_BODY_START=re.compile('{')

#----- FUNCTIONS -----

# Print error and exit.
def panic(msg):
        print "#>> Error when calling: %s" %(msg)
        sys.exit(1)

# Execute a command in a subprocess
def call(cmd):
        process = subprocess.Popen(cmd, shell=True, stdin=None, stdout=None, bufsize=0)
        r = process.wait()
        if r != 0:
```

```
40              panic(cmd)
41
42 # Inject the function call.
43 def inject_enable_function(FILE):
44       LOOK = False
45       DONE = False
46
47       F = open(FILE, "r")
48       TEXT = F.read()
49       F.close()
50
51       LINES=TEXT.split("\n")
52       F = open(FILE, "w")
53       F.write(RR_DEF + "\n")
54
55       for LINE in LINES:
56              F.write(LINE + "\n")
57
58              if not DONE:
59                     if LOOK:
60                            MO = RE_BODY_START.search(LINE)
61                            if MO != None and len(MO.group(0)) > 0:
62                                   F.write(RR_CALL + "\n")
63                                   DONE=True
64                     else:
65                            MO = RE_MAIN.search(LINE)
66                            if MO != None and len(MO.group(0)) > 0:
67                                   MO = RE_BODY_START.search(LINE)
68                                   if MO != None and len(MO.group(0)) > 0:
69                                          F.write(RR_CALL + "\n")
70                                          DONE = True
71                                   else:
72                                          LOOK = True
73       F.close()
74       print "Enabled rr in '%s'." %(FILE)
75
76 # Look for a main method and call inject function.
77 # If no main function was found and the file still contains the word "main" a warning is
       printed.
78 def check_file_for_main(FILE):
79       F = open(FILE,"r")
80       TEXT = F.read()
81       F.close()
82
83       MO=RE_MAIN.search(TEXT)
84       if MO != None and len(MO.group(0)) > 0:
85              inject_enable_function(FILE)
86       else:
87              MO = RE_MAIN_CHECK.search(FILE)
88              if MO != None and len(MO.group(0)) > 0:
```

```
                    print "WARNING: grep my have missed a main function in '%s'. Please
                        check manually." %(FILE)

# Find all relevant files.
def walk(X, DIR, FILES):
        for FILE in FILES:
                MO = RE_FILES.search(FILE)
                if MO != None and len(MO.group(0)) > 0:
                        check_file_for_main(DIR + "/" + FILE)

#----- SCRIPT -----

if len(ARGS) != 1:
        print "Usage: enable_rouding.py FOLDER"
        exit
else:
        FOLDER=ARGS[0]

os.path.walk(FOLDER, walk, "")
```

Listing A.4: enable_rounding.py - Automated Code Insertion enabler script

A.2.5 GROMACS 4.0.7 patch

```
diff -rupN ../gromacs-4.0.7-rr-single-clean/src/mdlib/ns.c ./src/mdlib/ns.c
--- ../gromacs-4.0.7-rr-single-clean/src/mdlib/ns.c 2010-09-21 23:57:27.000000000 +0200
+++ ./src/mdlib/ns.c  2010-08-26 10:41:54.000000000 +0200
@@ -83,6 +83,10 @@ static bool NOTEXCL_(t_excl e[],atom_id
 #define NOTEXCL(e,i,j) !(ISEXCL(e,i,j))
 #endif

+#define _EPSILON_ 0.0000001
+real _tmp_;
+real _tmp_ii_;
+
 /************************************************
  *
  *  U T I L I T I E S  F O R  N S
@@ -2039,7 +2043,15 @@ static int nsgrid_core(FILE *log,t_commr
                                                /* check energy group exclusions */
                                                if (!(i_egp_flags[jgid] & EGP_EXCL))
                                                {
-                                                   if (r2 < rs2)
+
     _tmp_ = r2-rs2;
+
     if(_tmp_ < 0) {
+
     _tmp_ = -1*_tmp_;
```

```
+
    }
+
            _tmp_ii_ = r2-rm2;
+                                                   if(_tmp_ii_ < 0) {
+                                                       _tmp_ii_ = -1*_tmp_ii_;
+                                                   }
+                                                   if (r2 < rs2 && _tmp_ > _EPSILON_)
                                                    {
                                                        if (nsr[jgid] >= MAX_CG)
                                                        {
@@ -2052,7 +2064,7 @@ static int nsgrid_core(FILE *log,t_commr
                                                        }
                                                        nl_sr[jgid][nsr[jgid]++]=jjcg;
                                                    }
-                                                   else if (r2 < rm2)
+                                                   else if (r2 < rm2 && _tmp_ii_ > _EPSILON_)
                                                    {
                                                        if (nlr_ljc[jgid] >= MAX_CG)
                                                        {
@@ -2129,14 +2141,22 @@ static int nsgrid_core(FILE *log,t_commr
     /* Perform any left over force calculations */
     for (nn=0; (nn<ngid); nn++)
     {
-        if (rm2 > rs2)
+       _tmp_ = rm2-rs2;
+       if(_tmp_ < 0) {
+           _tmp_ = -1*_tmp_;
+       }
+       if (rm2 > rs2 && _tmp_ > _EPSILON_)
        {
            do_longrange(cr,top,fr,0,md,icg,nn,nlr_ljc[nn],
                         nl_lr_ljc[nn],bexcl,shift,x,box_size,nrnb,
                         lambda,dvdlambda,grppener,
                         TRUE,TRUE,TRUE,bHaveVdW,bDoForces);
        }
-       if (rl2 > rm2) {
+              _tmp_ = rl2-rm2;
+       if(_tmp_ < 0) {
+           _tmp_ = -1*_tmp_;
+       }
+       if (rl2 > rm2 && _tmp_ > _EPSILON_) {
            do_longrange(cr,top,fr,0,md,icg,nn,nlr_one[nn],
                         nl_lr_one[nn],bexcl,shift,x,box_size,nrnb,
                         lambda,dvdlambda,grppener,
```

Listing A.5: gromacs-4.0.7.patch - GROMACS if-condition patch

A.2.6 System Configuration

The software and tools where developed and tested on this system configuration:
Software

- Linux 2.6.18-194.26.1.el5 SMP x86_64
- gcc version 4.1.2 20080704 (Red Hat 4.1.2-48)
- GNU libc 2.5
- GROMACS 4.0.7
- Python 2.4.3
- GNU bash, version 3.2.25(1)-release (x86_64-redhat-linux-gnu)
- grep (GNU grep) 2.5.1
- GNU sed version 4.1.5
- patch 2.5.4

Hardware

- Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz
- 12 Gb RAM
- 2x NVIDIA GeForce GTX 480

A.2.7 Compiling GROMACS with random rounding arithmetic

These are the steps necessary to compile GROMACS with random rounding arithmetic.

```
1 wget ftp://ftp.gromacs.org/pub/gromacs/gromacs-4.0.7.tar.gz
2 tar xzf gromacs-4.0.7.tar.gz
3 cd gromacs-4.0.7/
4 patch -Np1 < /tmp/gromacs-4.0.7.patch
5 /tmp/bin/enable_rounding.py /tmp/gromacs-4.0.7/
6 CC="/tmp/bin/gccrr.py" CFLAGS="-O0 -g" CPPFLAGS="-O0 -g" ./configure
      --prefix=/tmp/gromacs-rr/double --disable-x86-64-sse --disable-cpu-optimization
      --enable-double --disable-fftw-measure --with-fft=fftpack --disable-ia32-3dnow
      --disable-ia32-sse
7 make #(no make -j!!!)
8 make install
```

Listing A.6: Compiling GROMACS step-by-step

The patch from line four fixes an issue with some if conditions. Line five inserts a definition and the call of the initialization function for the random rounding arithmetic. Line six contains the configuration options. Optimization level O0 and debugging is really essential. This way the compiler does not re-arrange or optimize code so all arithmetic operations are translated straight forward. Disabling SSE[1] ensures that the arithmetic operations are translated to FPU commands which can be modified by the script from A.1. SSE accumulates several operations into one therefore preventing to set the rounding mode for each atomic operation individually. The current implementation of the compilation script does not allow several instances to run at any point in time. This is because the control word object file might be overwritten during the linking stage resulting in binary garbage. As a result the make command from line seven must not have a `-j` argument (or no number larger then 1 for `-j` to be absolutely correct). Be aware that GROMACS appends `_d` as suffix to all binary files if compiled with double precision.

A.2.8 Running the waterbox example

Starting the waterbox simulation involves two steps.

```
1 grompp -p 00\_spce.top -f sim.mdp -c 04\_md\_final.gro -o 05\_input.tpr
2 mdrun -reprod -v -s 05\_input.tpr -o 05\_md\_traj.trr -e 05\_md\_ener.edr -c
      05\_md\_final.pdb -g result.log
```

Listing A.7: Waterbox execution commands

The first command generates one input file which contains all data necessary to start the simulation. It is better to use `grompp` from an unmodified build of GROMACS and only start `mdrun` from the random rounding version. Line two contains the `-reprod` argument. It instructs GROMACS to eliminate functions which exacerbate the reproducibility whenever possible (see [gro]). All files named in line two are output files except for `05_input.tpr`. PyMol can use the `.pdb` file to visualize the result of the simulation (see chapter 4.2). To extract different properties from the simulation `g_energy` can be used on the energy file (`05_md_ener.edr`).

[1] http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitiv

A.3 GROMACS for GPU Related Works

A.3.1 Static Binary Rounding compiler script

The script replaces the original ptxas binary from CUDA and replaces the floating point operations with their corresponding round up or down version.

```
#!/usr/bin/python
#/home/moetzicn/python/bin/python
# -*- coding: utf-8 -*-

#----- IMPORT -----
import sys
from os import system
import os.path
import subprocess
import re
import random
import time

#----- SETTINGS -----
PTXAS = "ptxas_orig "        # Path to the unchanged ptxas binary.
DEBUG = True                        # Enable/disable debugging information.
RND = ["rm", "rp" ]                 # This are the two rounding modes the script will use.
LOG_FILE = "/tmp/ptxas"      # In in debug mode, write to this log.

# CUDA assembly fp instructions. Example for PTX ISA 2.2 and .target sm_20 only
# The script constructs all permutations e.g. mul.f32, mul.f64, mul.rz.f32, mul.rn.f32, ...
MATH_BASE = ["mul", "add", "sub" , "mad", "fma", "div", "rcp", "sqrt" ]
MATH_MID = [ "rz", "rn", "ftz", "sat" ]
MATH_EXT = [ "f32", "f64" ]

#----- INIT-----

# Regular expression to search for ptx input file.
PTX_CMD_REGEX=re.compile("\s+([^\s]+\.ptx)\s+")

CLEAN_ARGS = sys.argv[1:]
CMD_STR = " ".join(sys.argv)

#----- FUNCTIONS -----

# Return a random bit.
def get_rnb() :
        return random.randint(0,8192)%2


# Exits the script with an error if something bad happens.
def panic_code(msg, code):
```

```
43        global DEBUG
44        if DEBUG :
45                print "########################################"
46                print "#>> Panic: %s" %(msg)
47                print "########################################"
48        sys.exit(code)
49
50  # Exits the script silently.
51  def panic(msg):
52        panic_code(msg, 1)
53
54  # Execute given command
55  def execute(cmd):
56      if DEBUG :
57          print(cmd)
58      process = subprocess.Popen(cmd ,shell=True, stdin=None, stdout=None, bufsize=0)
59      r = process.wait()
60      if r != 0 :
61          panic("Could not execute command" + cmd)
62
63  # Randomly return round up or round down code.
64  def get_rounding_mode() :
65      return RND[get_rnb()]
66
67  # Replace all mathematical operator with rounding version.
68  def replace_math_op(file) :
69      fds = open(file, "r")
70      text = fds.read()
71      fds.close()
72      old_lines = text.split("\n")
73      new_lines = []
74      for line in old_lines :
75                rnd = get_rounding_mode()
76                for ext in MATH_EXT :
77                        match = re.search(ext, line)
78                        if match!=None :
79                                for base in MATH_BASE :
80                                        match=re.search("\s+" + base + "\.", line)
81                                        if match!=None :
82                                                if re.search("\s+" + base + "\." + ext, line) !=
                                                    None :
83                                                   new_op = base + "." + rnd + "." + ext
84                                                   op = base + "." + ext
85                                                   write_log("Replacing instruction " + op +
                                                        " with " + new_op)
86                                                   line = line.replace(op, new_op)
87                                                else :
88                                                   for mid in MATH_MID :
89                                                      if re.search("\s+" + base + "\." +
                                                           mid + "\." + ext, line) != None
                                                           :
```

```
90                                                              new_op = base + "." + rnd +
                                                                    "." + ext
91                                                              op = base + "." + mid + "."
                                                                    + ext
92                                                              write_log("Replacing
                                                                    instruction " + op + "
                                                                    with " + new_op)
93                                                              line = line.replace(op,
                                                                    new_op)
94                                                              break
95                              break
96              new_lines.append(line)
97      fds = open(file, "w")
98      fds.write("\n".join(new_lines))
99      fds.close()
100
101 # Get the path of the ptx file.
102 def get_ptx_file(args) :
103         match=PTX_CMD_REGEX.search(args)
104         if match != None and len(match.group(1)) > 0:
105                 return match.group(1)
106         else :
107                 return None
108
109 # If in debug mode, write the message to the log file.
110 def write_log(message) :
111         if DEBUG :
112                 fds = open(LOG_FILE, "a")
113                 fds.write(message + "\n")
114                 fds.close()
115
116 #----- SCRIPT -----
117 random.seed(time.time())
118
119 write_log("Original commad: " + CMD_STR)
120 ptx_file = get_ptx_file(CMD_STR)
121 if ptx_file != None :
122         write_log("Replacing fp operations in " + ptx_file)
123         replace_math_op(ptx_file)
124 else :
125         if DEBUG :
126                 print "Could not find a ptx file in command:\n" + CMD_STR
127
128 # Execute the unchanged binary with the original arguments.
129 execute("ptxas_orig " + " ".join(CLEAN_ARGS))
```

Listing A.8: ptxas.py - Static Binary Rounding compiler script

A.3.2 Compiling GROMACS for GPU with random rounding arithmetic

These are the commands necessary to compile OpenMM and GROMACS with random rounding arithmetic on GPU. The source code of OpenMM can be obtained from https://simtk.org/project/xml/downloads.xml?group_id=161. After registering one needs to download OpenMM2.0-Source.zip. The floating point operations replaced by the script only work with compute capability 2.0 or newer (see [NVI10a] and [NVI10b]). Therefore all other compute capability targets need to be eliminated. The floating point operations are replaced before the ptx assembly phase of `nvcc` (see figure 5.5). Therefore the binary file `ptxas` in the CUDA bin folder needs to replaced with the python script from appendix A.3.1. The original `ptxas` command needs to be renamed to `ptxas_orig` so that the script can still execute it after it has done its modifications. It is not sufficient to only put the python script somewhere in the PATH variable of the system since CMake tries to determine the location of CUDA via the location of the CUDA binaries.

```
1  unzip OpenMM2.0-Source.zip
2  wget ftp://ftp.gromacs.org/pub/gromacs/gromacs-4.5.3.tar.gz
3  tar xzf gromacs-4.5.3.tar.gz
4  mkdir openmm
5  cd openmm
6  cmake -DCMAKE_INSTALL_PREFIX=/tmp/openmm -DFOUND_CUDART=/tmp/cuda/lib64/libcudart.so
       -DCUDA_HAVE_GPU=TRUE -DOPENMM_BUILD_CUDA_LIB=ON -DOPENMM_BUILD_OPENCL_LIB=OFF
       -DOPENMM_BUILD_STATIC_LIB=ON -DFOUND_CUT_INCLUDE=/tmp/cuda/sdk/C/common/inc
       -DFOUND_CUT=/tmp/cuda/sdk/C/lib/libcutil_x86_64.a ../OpenMM2.0-Source/src/
7  sed -i "s/CUDA_NVCC_FLAGS:STRING=-gencode;arch=compute_11,code=sm_11;
       -gencode;arch=compute_13,code=sm_13;-gencode;arch=compute_20,code=sm_20;
       -use_fast_math/CUDA_NVCC_FLAGS:STRING=
       -gencode;arch=compute_20,code=sm_20;--ftz=false;--prec-div=true;--prec-sqrt=true;/g"
       CMakeCache.txt
8  echo "c\ng\n" > /tmp/cmake.cnf
9  (ccmake CMakeCache.txt) < /tmp/cmake.cnf
10 make
11 make install
12 cd ..
13 mkdir gromacs
14 cd gromacs
15 export OPENMM_ROOT_DIR="/tmp/openmm"
16 cmake -DGMX_OPENMM=ON -DCMAKE_INSTALL_PREFIX=/tmp/gromacs-gpu -DGMX_THREADS=OFF
       ../gromacs-4.5.3
17 make mdrun
18 make install-mdrun
```

Listing A.9: Compile GROMACS for GPU step-by-step

Bibliography

[AAB+10] E. Apol, R. Apostolov, H. J. Berendsen, A. van Buuren, P. Bjelkmar, R. van Drunen, A. Feenstra, G. Groenhof, P. Kasson, P. Larsson, P. Meulenhoff, T. Murtola, S. Pall, S. Pronk, R. Schulz, M. Shirts, A. Sijbers, P. Tieleman, B. Hess, D. van der Spoel, E. Lindahl. *GROMACS User Manual*. Department of Biophysical Chemistry, University of Groningen, version 4.5 edition, 2010. (Cited on page 30)

[All04] M. P. Allen. Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins*, 23:1–28, 2004. (Cited on page 30)

[Avr08] V. Avrutin. Einführung in die Chaostheorie. Lecture Notes, 2008. (Cited on page 32)

[CFD08] S. Collange, J. Flórez, D. Defou. A GPU interval library based on Boost interval. 2008. (Cited on page 25)

[Che90] J.-M. Chesneaux. Study of the computing accuracy by using probabilistic approach. *Contribution to computer arithmetic and self-validating numerical methods*, pp. 19–30, 1990. (Cited on page 17)

[CK] M. J. Corden, D. Kreitzer. Consistency of Floating-Point Results using the Intel® Compiler. URL http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/. (Cited on page 16)

[CMa] R. Chotin, H. Mehrez. A Floating-Point Unit Using Stochastic Arithmetic Compliant With The IEEE-754 Standard. URL http://www-asim.lip6.fr/pub/reports/2002/ar.chot.icecs02.pdf. (Cited on page 21)

[CMb] R. Chotin, H. Mehrez. Hardware implementation of a method to control round-off errors. URL http://www-asim.lip6.fr/pub/reports/2002/ar.chot.cscc02.pdf. (Cited on page 21)

[CT] Cadna-Team. CADNA for C/C++ source codes. (Cited on page 18)

[fen] ISO C99 - fenv.h. URL http://www.opengroup.org/onlinepubs/000095399/basedefs/fenv.h.html. (Cited on page 16)

[gro] GROMACS - Reproducibility. URL http://www.gromacs.org/Documentation/Terminology/Reproducibility. (Cited on pages 16 and 76)

[HL] J. C. Huang, T. Leng. Generalized Loop-Unrolling: a Method for Program Speed-Up. (Cited on page 54)

[Int11] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Arithmetic*. Intel, 2011. (Cited on page 22)

[MSW95] H. L. Muller, P. W. Stallard, D. H. Warren. Hiding Miss Latencies with Multithreading on the Data Diffusion Machine. *International Conference on Parallel Processing*, 1:178–185, 1995. (Cited on page 47)

[NVI10a] NVIDIA. NVIDIA CUDA C Programming Guide, 2010. URL http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf. (Cited on page 80)

[NVI10b] NVIDIA. PTX: Parallel Thread Execution ISA Version 2.1, 2010. URL http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf. (Cited on page 80)

[NVI10c] NVIDIA. PTX: Parallel Thread Execution ISA Version 2.2, 2010. (Cited on page 49)

[NVI10d] NVIDIA. Tuning CUDA^{TM} Applications for Fermi^{TM}, 2010. (Cited on page 51)

[PV80] M. L. Porte, J. Vignes. Algorithmes numériques, analyse et mise en oeuvre. *Edition Technip*, 1980. (Cited on page 15)

[RCC+06] R. Ramanathan, R. Curry, S. Chennupaty, R. L. Cross, S. Kuo, M. J. Buxton. Extending the World's Most Popular Processor Architecture, 2006. URL ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf. (Cited on page 33)

[Rüo8] V. Rühle. Pressure coupling / barostats. 2008. (Cited on page 30)

[Vig78] J. Vignes. New methods for evaluating the validity of the results of mathematical computation. *Mathematics and Computers in Simulation*, 20:227–249, 1978. (Cited on page 15)

[Vig88] J. Vignes. Review on stochastic approach to round-off error analysis and its applications. *Mathematics and Computers in Simulation*, 30:481–491, 1988. (Cited on page 15)

[Vig93] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35:233–261, 1993. (Cited on pages 15, 17, 18 and 38)

All links were last followed on January 30, 2011.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Christian Mötzing)