Bose-Einstein condensation in trapped bosons: A quantum Monte Carlo analysis using OpenCL and GPU programming

by

IVAR URSIN NIKOLAISEN

THESIS

for the degree of

MASTER OF SCIENCE

(Master in Computational Physics)

Faculty of Mathematics and Natural Sciences
Department of Physics
University of Oslo

June 2011

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Contents

Chapter 1

Introduction

The aim of this thesis is to develop a diffusion Monte Carlo (DMC) and a Variational Monte Carlo (VMC) program written in OpenCL in order to test the potential for using Graphical Processing Units (GPUs) in Monte Carlo studies of both bosonic and fermionic systems. The main interest is in whether such an OpenCL implementation can offer competitive performance compared to implementations in C/C++. The hope is that modern GPUs prove suitable, in that they can show better performance both by price and power consumption compared to systems using CPUs, and that the difficulty in programming them is not too great. It is also of interest whether the OpenCL implementation performs well on CPUs (looking for the always elusive "code once, run everywhere" property).

The possibility of moving numerical simulations from the CPU to the GPU has arisen mostly because of the large market for graphics cards for computer games. It is particularly after the year 2006 that GPUs have become truly suited for general purpose tasks, as there was a significant shift in the industry at that time with the introduction of Microsoft's DirectX 10, and the associated move to more flexible GPU architectures by the hardware manufacturers.

Programming GPUs has been done in many ways, originally requiring that the problem be mapped to a graphics-like algorithm and implemented using graphics-oriented approaches like DirectX or OpenGL. In the last few years however three prominent methods to do general purpose programming for GPUs have surfaced. One is CUDA (Compute Unified Device Architecture), which is specific to Nvidia GPUs. Another is DirectCompute, which is part of Microsoft's DirectX and runs only on the Microsoft Windows operating system, but supports most modern GPUs. The third and most broadly supported is OpenCL, which is an open standard managed by the Khronos Group. Implementations are provided by many parties, such as AMD, Nvidia, Intel and IBM. It has the advantage of not being specific to any manufacturer or operating system.

In this thesis I have used the OpenCL framework to implement VMC and DMC solvers for two different systems of interest in physics. One is a simulation of the atoms helium, beryllium and neon using only VMC, and the other

deals with Bose-Einstein condensation (BEC) in gases of alkali atoms like ^{87}Rb confined in a magnetic trap that can be approximated by a harmonic oscillator potential. This system will be simulated using both VMC and DMC. This BEC was first experimentally demonstrated by Anderson *et al.* [21]. The extensive progress made in the study of such BECs up to early 1999 is reviewed by Dalfovo *et al.* [14].

The systems to be studied have already been numerically simulated and studied by others [13, 24, 25, 28, 29]. Therefore this thesis does not aim to introduce any new physics or results, but rather focuses on implementing VMC and DMC solvers in OpenCL. Results and performance will then be compared to other relevant work.

The program is in principle extendable, and the general approach taken should be suitable for a broad range of Monte Carlo simulations of this type with only minor modifications. The program is meant to run well on both CPUs and GPUs, and relies on standard OpenCL with few architecture-specific optimizations. As such the benchmarks presented can be seen as a comparison between what can be done with relatively straight-forward C/C++ versus OpenCL.

Since pseudo-random numbers are important in Monte Carlo simulations like the ones undertaken in this thesis, and since there is still not a large collection of pseudo-random number generators available for OpenCL I have also implemented the RANLUX [22] pseudo-random number generator. The generator has interesting properties, and is well suited for use on GPUs.

The thesis is organized as follows. Chapter 2 presents fundamental quantum mechanics, however the reader that is already familiar with the topic or is more interested in the algorithms and implementations can safely skip it.

In chapter 3 the basics of the Monte Carlo methods used are presented. We look at basic Monte Carlo integration, Markov chains, and state the results needed to implement VMC and DMC with importance sampling. We also look at the tools needed to perform statistical analyses on the generated results, namely the so-called blocking analysis method. The reader merely interested in the implementation and results can skip most of it, perhaps just viewing algorithms 3.1 and 3.2 outlining the VMC and DMC methods.

In chapter 4 the systems, namely the atoms and the BEC are introduced. We look at the Hamiltonian and trial wave functions we will use.

In chapter 5 the basics of OpenCL and GPGPU are introduced, providing a rough overview of what OpenCL is and of the general flow of an OpenCL application. It can safely be skipped by readers familiar with OpenCL and GPU programming.

In chapter 6 the actual OpenCL implementation of the VMC and DMC solvers is presented. This includes descriptions of the implementation of RANLUX, the closed-form derivatives needed for our different wave functions, and the general layout of the program. Most of the OpenCL C kernel code is listed with descriptions.

In chapter 7 the results are presented. The generated results are compared to known literature, and the performance of the implementation is also studied.

Lastly in chapter 8 the work is summed up, and possible improvements to the implementation are discussed.

Chapter 2

Fundamental Quantum Mechanics

2.1 The Postulates

Nonrelativistic Quantum Mechanics has four postulates. See for example page 115 in ref. [27].

	Classical Mechanics	**Quantum Mechanics**
I	The state of a particle at any given time is specified by the two variables $x(t)$ and $p(t)$, i.e., as a point in a two-dimensional phase space.	The state of the particle is represented by a vector $\|\psi(t)\rangle$ in a Hilbert space.
II	Every dynamical variable ω is a function of x and p: $\omega(x,\ p)$.	The independent variables x and p of classical mechanics are represented by Hermitian operators X and P with the following matrix elements in the eigenbasis of X $\langle x\|X\|x'\rangle = x\delta(x-x')$ $\langle x\|P\|x'\rangle = -i\hbar\delta'(x-x')$ The operators corresponding to dependent variables $\omega(x,\ p)$ are Hermitian operators given by $\Omega(X,\ P) = \omega(x \to X,\ p \to P)$.

III	If the particle is in a state given by x and p, the measurement of the variable ω will yield a value $\omega(x,\ p)$. The state will remain unaffected.	If the particle is in a state $\|\psi\rangle$, measurement of the variable (corresponding to) Ω will yield one of the eigenvalues ω with probability $P(\omega) \propto \|\langle\omega\|\psi\rangle\|^2$. The state of the system will change from $\|\psi\rangle$ to $\|\omega\rangle$ as a result of the measurement.
IV	The state variables change with time according to Hamilton's equations: $\dot{x} = \frac{\partial \mathcal{H}}{\partial p}$ $\dot{p} = -\frac{\partial \mathcal{H}}{\partial x}$	The state vector $\|\psi(t)\rangle$ obeys the Schrödinger equation $i\hbar\frac{d}{dt}\|\psi(t)\rangle = H\|\psi(t)\rangle$ where $H(X,\ P) = \mathcal{H}(x \to X,\ p \to P)$ is the quantum Hamiltonian operator and $\mathcal{H}$ is the Hamiltonian for the corresponding classical problem.

The postulates (of classical and quantum mechanics) fall naturally into two sets: the first three tell us how the system is depicted at a given time, and the fourth specifies how this picture changes with time.

The first postulate states that a particle is described by a ket $|\psi\rangle$ in a Hilbert space, which means that the ket has in general an infinite number of components in a given basis. The "reason" for this lies in the next two postulates, where we see that the ket represents a probability amplitude

When it is said that $|\psi\rangle$ is an element of a vector space we mean that if $|\psi\rangle$ and $|\psi'\rangle$ represent possible states, then $\alpha|\psi\rangle + \beta|\psi'\rangle$ is also a possible state. This is the principle of superposition.

Postulates II and III state that for a classical system in a known state $(x,\ p)$, one can say that any dynamical variable ω has a value $\omega(x,\ p)$, meaning that if ω is measured it is know what the result will be. In the case of quantum physics the result is instead given by:

1. Construct the corresponding quantum operator $\Omega = \omega(x \to X,\ p \to P)$, where X and P are the operators defined in postulate II.
2. Find the orthonormal eigenvectors $|\omega_i\rangle$ and eigenvalues ω_i of Ω.
3. Expand $|\psi\rangle$ in this basis: $|\psi\rangle = \sum_i |\omega_i\rangle\langle\omega_i|\psi\rangle$.
4. The probability $P(\omega)$ that the result ω will be obtained is proportional to the modulus squared of the projection of $|\psi\rangle$ along the eigenvector $|\omega\rangle$, that is $P(\omega) \propto |\langle\omega|\psi\rangle|^2$. In terms of the projection operator $P_\omega = |\omega\rangle\langle\omega|$, $P(\omega) \propto |\langle\omega|\psi\rangle|^2 = \langle\psi|\omega\rangle\langle\omega|\psi\rangle = \langle\psi|P_\omega|\psi\rangle = \langle\psi|P_\omega P_\omega|\psi\rangle = \langle P_\omega\psi|P_\omega\psi\rangle$.

The theory only makes probabilistic predictions for the result of a measurement of Ω. It also only assigns relative probabilities to the different eigenvalues of Ω,

and since postulate II says that Ω is Hermitian, all the possible results are the eigenvalues of Ω, which are all real.

Note that since $P(\omega_i) \propto |\langle \omega_i | \psi \rangle|^2$, the quantity $|\langle \omega_i | \psi \rangle|^2$ is only the relative probability. To get the absolute probability, we divide by the sum of all relative probabilities:

$$P(\omega_i) = \frac{|\langle \omega_i | \psi \rangle|^2}{\sum_j |\langle \omega_j | \psi \rangle|^2} = \frac{|\langle \omega_i | \psi \rangle|^2}{\langle \psi | \psi \rangle}.$$

If the starting state had been a normalized state $|\psi'\rangle$ where

$$|\psi'\rangle = \frac{|\psi\rangle}{\langle \psi | \psi \rangle^{1/2}},$$

the result would be:

$$P(\omega_i) = |\langle \omega_i | \psi' \rangle|^2 .$$

Also note that if $|\psi\rangle$ is an eigenstate $|\omega_i\rangle$ of Ω the measurement is guaranteed to yield the result ω_i.

2.2 The Schrödinger Equation

The final postulate describes how the system evolves in time, governed by the Schrödinger equation

$$i\hbar \frac{d}{dt} |\psi(t)\rangle = H |\psi(t)\rangle . \tag{2.1}$$

Again following [27], we divide the discussion into three sections:

1. Setting up the equation
2. General approach to its solution
3. Choosing a basis for solving the equation

2.2.1 Setting Up the Equation

To set up the equation one must simply make the substitution $\mathcal{H}(x \to X,\ p \to P)$, where $\mathcal{H}$ is the classical Hamiltonian for the same problem. Thus, if we are describing a harmonic oscillator, which is classically described by the Hamiltonian

$$\mathcal{H} = \frac{p^2}{2m} + \frac{1}{2} m\omega^2 x^2,$$

the Hamiltonian in quantum mechanics becomes

$$H = \frac{P^2}{2m} + \frac{1}{2} m\omega^2 X^2, \tag{2.2}$$

and in three dimensions

$$H = \frac{P_x^2 + P_y^2 + P_z^2}{2m} + \frac{1}{2}m\omega^2\left(X^2 + Y^2 + Z^2\right).$$

2.2.2 General Approach to the Solution

Let us first assume that H has no explicit time dependence. Our approach is to find the eigenvectors and eigenvalues of H and to construct the propagator $U(t)$ in terms of these. We can then write

$$|\psi(t)\rangle = U(t)\,|\psi(0)\rangle.$$

Let us now construct an explicit expression for $U(t)$ in terms of $|E\rangle$, the normalized eigenstates of H with eigenvalues E which obey

$$H\,|E\rangle = E\,|E\rangle.$$

This is the time-independent Schrödinger equation. Assume that we have solved it and found the eigenstates $|E\rangle$. If we expand $|\psi\rangle$ as

$$|\psi(t)\rangle = \sum |E\rangle\langle E|\psi(t)\rangle \equiv \sum a_E(t)\,|E\rangle,$$

the equation for $a_E(t)$ follows if we act on both sides with $i\hbar\frac{\partial}{\partial t} - H$

$$0 = \left(i\hbar\frac{\partial}{\partial t} - H\right)|\psi(t)\rangle = \sum\left(i\hbar\dot{a}_E - Ea_E\right)|E\rangle \Rightarrow i\hbar\dot{a}_E = Ea_E,$$

where we have used the fact that the vectors $|E\rangle$ are linearly independent. The solution is

$$a_E(t) = a_E(0)\,e^{-iEt/\hbar},$$

or

$$\langle E|\psi(t)\rangle = \langle E|\psi(0)\rangle e^{-iEt/\hbar},$$

so that

$$|\psi(t)\rangle = \sum_E |E\rangle\langle E|\psi(0)\rangle e^{-iEt/\hbar}.$$

We can now extract $U(t)$

$$U(t) = \sum_E |E\rangle\langle E|\,e^{-iEt/\hbar}.$$

2.2.3 Choosing a Basis for Solving the Equation

Barring a few exceptions, the Schrödinger equation is always solved in a particular basis. Although all bases are equal mathematically, "some are more equal than others". First of all, since $H = H(X,\ P)$ the X and P bases recommend themselves, for in going to one of them the corresponding operator is rendered diagonal. Thus one can go to the X basis in which $X \to x$ and $P \to -i\hbar\frac{d}{dx}$ or to the P basis in which $P \to p$ and $X \to i\hbar\frac{d}{dp}$. The choice between the two depends on the Hamiltonian. Assuming it is of the form (in one dimension)

$$H = T + V = \frac{P^2}{2m} + V(X),$$

the choice is dictated by $V(X)$. Since $V(X)$ is usually a more complicated function of X than T is of P, one prefers the X basis. Thus if

$$H = \frac{P^2}{2m} + \frac{1}{\cosh^2(X)},$$

the equation

$$H|E\rangle = E|E\rangle,$$

becomes in the X basis the second-order equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{\cosh^2(x)}\right)\psi_E(x) = E\psi_E(x),$$

which we can find a closed form solution for. Had we used the P basis, we would instead find

$$\left[\frac{p^2}{2m} + \frac{1}{\cosh^2\left(i\hbar\frac{d}{dp}\right)}\right]\psi_E(p) = E\psi_E(p),$$

which would be much more difficult to handle.

2.3 The Harmonic Oscillator

This subsection will introduce the harmonic oscillator, both because it is generally very relevant as a quantum mechanical system, and because the Bose-Einstein condensate that will be numerically simulated later is trapped in a potential which can be approximated by a harmonic oscillator potential.

The harmonic oscillator is a central system both in classical and quantum mechanics, and it can be solved exactly in both cases. It shows up in many applications because any system fluctuating around a stable equilibrium may be described by a harmonic oscillator or by a collection of oscillators.

2.3.1 The Classical Oscillator

The common example of a classical harmonic oscillator is that of a mass on a spring, which consists of a mass m coupled to a spring with force constant k. Using Hooke's law the force of the spring is given by $F = -kx^2$, which produces a potential $V(x) = \frac{1}{2}kx^2$. The Hamiltonian of this system is then

$$\mathcal{H} = T + V = \frac{p^2}{2m} + \frac{1}{2}m\omega^2 x^2,$$

where $\omega = \sqrt{k/m}$ is the frequency of the oscillator.

The equations of motion of the classical harmonic oscillator are (see eq. (7.2.1-2) in ref. [27])

$$\begin{aligned} \dot{x} &= \frac{\partial \mathcal{H}}{\partial p} = \frac{p}{m} \\ \dot{p} &= -\frac{\partial \mathcal{H}}{\partial x} = -m\omega^2 x. \end{aligned}$$

By eliminating $\dot{p}$ we get the familiar equation

$$\ddot{x} + \omega^2 x = 0,$$

with the solution (eq. (7.2.3) in ref. [27])

$$x(t) = A\cos\omega t + B\sin\omega t = x_0\cos(\omega t + \phi),$$

where x_0 is the amplitude and ϕ the phase of the oscillator. The conserved energy of the oscillator is

$$E = T + V = \frac{1}{2}m\dot{x}^2 + \frac{1}{2}m\omega^2 x^2 = \frac{1}{2}m\omega^2 x_0^2.$$

Since x_0 is a continuous variable the energy is so too. The lowest value is $E = 0$, which corresponds to the particle remaining at rest at the bottom of the potential.

2.3.2 The Quantum Oscillator

As follows from postulate IV, the quantum mechanical oscillator is a particle whose state vector $|\psi\rangle$ obeys the Schrödinger equation (2.1), with the Hamiltonian given by the substitution

$$H = \mathcal{H}(x \to X, p \to P) = \frac{P}{2m} + \frac{1}{2}m\omega^2 X^2.$$

Going to the x-basis where $X \to x$ and $P \to -i\hbar\frac{d}{dx}$ the Hamiltonian becomes

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega^2 x^2.$$

The ground state wave function is (eq. (7.3.22) in ref. [27])

$$\psi_0(x) = \exp\left(-\frac{m\omega x^2}{2\hbar}\right), \tag{2.3}$$

where the normalization term has been dropped. The energies are given by (eq. (7.3.20) in ref. [27])

$$E_n = \left(n + \frac{1}{2}\right)\hbar\omega,$$

where $n \geq 0$ is an integer, and $n = 0$ gives us the ground state energy $E_0 = \frac{1}{2}\hbar\omega$. We see that the quantum oscillator can only take discrete energies, as opposed to the classical case where the energy is continuous.

2.4 Identical Particles

We will be dealing with both bosons and fermions. This section aims to establish how we deal with (and what we mean by) identical particles, and what the difference is between a boson and a fermion.

We say that two particles are identical if they are exact replicas of each other in every respect, i.e. there should be no experiment that detects any intrinsic difference between them. While this definition is the same for identical particles both in classical and quantum mechanics, the implications are quite different.

2.4.1 The Classical Case

In classical physics we can have identical particles. But when we exchange the positions of two particles, we consider the new state distinct from the previous. Intuitively this is because we can assign labels to the particles even if they are identical.

If we don't allow ourselves the luxury of painting the particles with different colors (they wouldn't technically be identical anymore), we could for instance recruit undergraduate students to keep track of which particle has which label. In general there is nothing in classical physics that keeps us from keeping track of which particle is which, even when the particles are otherwise indistinguishable from one another. In other words the property that will never be identical for two particles is their histories.

2.4.2 Two-Particle Systems - Symmetric and Antisymmetric States

Suppose we have a system of two distinguishable particles 1 and 2 and a position measurement of the system shows that particle 1 is at $x = a$ and particle 2 is at $x = b$. We write this state as

$$|\psi\rangle = |x_1 = a,\ x_2 = b\rangle = |ab\rangle\,,$$

where we use the convention that the first label corresponds to the first particle and so on. Since the particles are distinguishable, the state obtained by exchanging them is distinguishable from the above, i.e. $|ab\rangle \neq |ba\rangle$.

Suppose we repeat the experiment with two identical particles. Again we find the particles at positions a and b, but is the state $|ab\rangle$ or $|ba\rangle$? The answer is that it is neither. We need a mixture of the two states that makes no reference to which particle is in which position. The two possibilities are

$$|ab,\ S\rangle = |ab\rangle + |ba\rangle\,,$$

called the symmetric state, and

$$|ab,\ A\rangle = |ab\rangle - |ba\rangle\,,$$

called the antisymmetric state. We note that in the antisymmetric state, switching the positions of two particles changes the sign

$$|ab,\ A\rangle = -\,|ba,\ A\rangle\,.$$

2.4.3 Bosons and Fermions

It turns out that all particles must "choose" whether they form symmetric or antisymmetric states. We call the particles that fall into symmetric states bosons, and particles that fall into antisymmetric states fermions. Examples of bosons are photons, pions and gravitons. Examples of fermions include electrons, protons and neutrons. More complicated particles, for example an atom, can also be bosons, even though it is comprised of fermions (electrons and nucleons).

We can now consider what will happen if two identical particles are trying to occupy the same state $a = b$. For bosons we have

$$|aa,\ S\rangle = |aa\rangle + |aa\rangle = 2\,|aa\rangle\,,$$

And there is no problem. For fermions however we get

$$|aa,\ A\rangle = |aa\rangle - |aa\rangle = 0.$$

This is the Pauli exclusion principle: Two identical fermions cannot be in the same quantum state.

It turns out that whether a particle is a boson or a fermion is connected to the spin of the particle. The magnitude of spin of a type of particle is invariant, and can only be one of the following values: 0, $\hbar/2$, $\hbar$, $3\hbar/2$, $2\hbar$, ... Particles with magnitude of spin equal to an even multiple of $\hbar/2$ are bosons, while particles with an odd multiple are fermions.

Usually spin is given as a pure number (i.e. dropping the $\hbar$). This means that bosons may have spins like 0, 1, 2 ... while fermions have spins like 1/2, 3/2, 5/2 and so on.

2.4.4 Dealing With Fermions

As we have seen bosons can have several particles occupying the same state, which means no special attention is needed. However for fermions we need a way to construct a wave function that does not allow two particles to occupy the same state, i.e. the wave function must be antisymmetric.

This can be accomplished by constructing the wave function as a so-called Slater determinant (see section 14.5 and 15.2 in ref. [23]). With $\{\phi_i\}$ being N orthonormal orbitals the Slater determinant is constructed as

$$\Psi\left(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\right) = \frac{1}{\sqrt{N}} \begin{vmatrix} \phi_1\left(\mathbf{x}_1\right) & \phi_1\left(\mathbf{x}_2\right) & \cdots & \phi_1\left(\mathbf{x}_N\right) \\ \phi_2\left(\mathbf{x}_1\right) & \phi_2\left(\mathbf{x}_2\right) & \cdots & \phi_2\left(\mathbf{x}_N\right) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_N\left(\mathbf{x}_1\right) & \phi_N\left(\mathbf{x}_2\right) & \cdots & \phi_N\left(\mathbf{x}_N\right) \end{vmatrix}.$$

This has the property that Ψ is antisymmetric

$$\Psi\left(\mathbf{x}_1, ..., \mathbf{x}_i, ..., \mathbf{x}_j, ..., \mathbf{x}_N\right) = -\Psi\left(\mathbf{x}_1, ..., \mathbf{x}_j, ..., \mathbf{x}_i, ..., \mathbf{x}_N\right).$$

Chapter 3

Statistics and Monte Carlo

This section provides an overview of the Monte Carlo methods used in this thesis, namely Variational Monte Carlo (VMC) and Diffusion Monte Carlo (DMC). Monte Carlo is a term used for almost any method where random numbers play an important role.

3.1 Random Numbers

When we talk about random numbers in the context of computer simulations, we usually really mean pseudo-random numbers. A pseudo-random sequence should appear random, but it is actually based on a deterministic algorithm and as such is not really random at all.

The three most central concepts are

- Random variables
- Probability Distribution Functions (PDF)
- Moments of a PDF

The typical example of a random distribution is that of a normal six sided die. In the case of a fair die, the possible outcomes are given by the set $[1, 2, 3, 4, 5, 6]$, which we will call the domain of our random variable. The domain describes what values the random variable can take.

We also have the PDF $[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$, which gives the probability of each of the possible outcomes.

We will denote a random variable by X, and the domain by D. This means that X can only take values that are in D: $X \in D$.

We will denote the PDF as $p(x) = Prob\,(X = x)$.

For the discrete case of the die these definitions are quite obvious, but there are also continuous distributions, which is what we will be using. Perhaps the most common continuous distribution is the uniform distribution of real numbers between 0 and 1, end points excluded. The PDF of this distribution is

$$p(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & else \end{cases}$$

In the continuous case the PDF no longer gives the probability of a specific value. Instead the probability for the variable X to take any value on an infinitesimal interval around x is given by $p\left(x\right) dx$. Thus the probability of getting a value between some numbers a and b (i.e. to have $a \leq X \leq b$) is $\int_a^b p(x)dx$.

A PDF must also be normalized, i.e. the total probability must be 1.

$$1 = \int_{x \in D} p\left(x\right) dx.$$

The expectation value of some function f with respect to a PDF is given as

$$\langle f \rangle = \int f\left(x\right) p\left(x\right) dx.$$

A special and important class of expectation values are the the moments of a PDF. The n-th moment is given by

$$\langle x^n \rangle = \int x^n p\left(x\right) dx.$$

The first moment is the mean value of the PDF (i.e. the average of a large sample of values X drawn from the PDF)

$$\mu = \langle x \rangle = \int x p\left(x\right) dx,$$

also referred to as the expectation value of p.

A special case of moments are the central moments, where the n-th central moment is given as

$$\langle (x - \langle x \rangle)^n \rangle = \int \left(x - \langle x \rangle\right)^n p\left(x\right) dx.$$

The second central moment is of particular interest, it is known as the variance of p

$$\begin{aligned} \sigma^2 &= \left\langle \left(x - \langle x \rangle\right)^2 \right\rangle \\ &= \int \left(x - \langle x \rangle\right)^2 p\left(x\right) dx \\ &= \left\langle x^2 \right\rangle - \langle x \rangle^2 . \end{aligned}$$

The square root of the variance is the standard deviation

$$\sigma = \sqrt{\left\langle \left(x - \langle x \rangle\right)^2 \right\rangle}.$$

3.2 Error Estimation

When we solve the systems in this thesis, we will be generating values (most importantly energies) in a stochastic process, producing a chain of values

$$\{x_1, x_2, ..., x_k, ...\}.$$

We'll denote an individual value a measurement and the set of all these values a sample. We assume these values are distributed according to some PDF, but all we will be interested in are the first and second moments (i.e. the sample mean μ and sample variance σ^2). The sample mean is then

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

while the sample variance is

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_k - \bar{x}_n)^2,$$

and the standard deviation of the sample is σ (the square root of the variance). Another interesting quantity is the sample covariance (see page 202 in ref. [1])

$$\mathrm{Cov}\,(x) = \frac{1}{n}\sum_{ij} (x_i - \bar{x}_n)(x_j - \bar{x}_n).$$

The sample covariance is a measure of the correlation between different measurements in the sample. If the measurements are uncorrelated the elements where $i \neq j$ sum up to zero, and the sample covariance is the same as the variance.

As the number of samples n increases, we expect the sample mean to converge to the true mean of the PDF we are sampling. However we also want to know how good of an estimate our sample mean is. We want to find the error of our result. This is given by the standard deviation of the sample mean, which is not the same as the standard deviation of the sample.

Say we generate a sample with 1000 measurements, and calculate the sample mean. One way to find the error would be to generate 100 samples with 1000 measurements each, calculate the sample means and then find the standard deviation of those 100 sample means. While this could give us a good estimate of the error of a sample, we would much rather combine all of the measurements into one big sample with 100000 measurements. But then we again face the problem of finding the error of the sample mean of our big sample.

Therefore we want a better way to find the error. A good but potentially slow way is through the sample covariance (see eq. (60) of ref. [1])

$$\mathrm{err}_x = \sqrt{\frac{1}{n}\mathrm{Cov}\,(x)},$$

however since the sample covariance has $\mathrm{O}\left(n^2\right)$ time complexity this evaluation can get very difficult when we have many measurements. Note that if the measurements are not correlated the above expression becomes

$$\mathrm{err_x} = \frac{\sigma}{\sqrt{\mathrm{n}}}, \tag{3.1}$$

which is very easy to compute. However in our simulations there will often be correlations, so this expression will only be useful if we can be reasonably certain the measurements are uncorrelated. The reason we have correlations is that we are using a Markov chain (discussed in section 3.4), and so each step is going to be correlated with preceding steps.

Another way to estimate the error is through what is called blocking analysis (see section 1.3.5 in ref. [19]). The idea is to partition the data into blocks of values, treating the mean value of each block as a measurement. We then calculate the error of the sample (using the means of the blocks as the measurements) as if though the data were uncorrelated (i.e. by eq. (3.1)). As the block size is increased, the estimated error will hopefully reach a plateau. This plateau of estimated errors is the estimate for the standard error of the full data set. Qualitatively we can say that this plateau will be reached when the block size is similar to the longest distance between correlated values.

A function for doing blocking analysis on a data set is included in the implementation (see ref. [6]), in the file *iunBlocking.hpp*. Blocking analysis is the method used to estimate the errors in the results in chapter 7.

3.3 Basic Monte Carlo Integration

The first natural example of the Monte Carlo method is that of integrating some function f on an interval

$$I = \int_a^b f(x)dx.$$

The general way to solve such an integral numerically is through the sum

$$I = \frac{(b-a)}{N} \sum_{i=1}^{N} w_i f\left(x_i\right),$$

where N is the number of integration points, w_i are weights to be determined and x_i are the integration points. All integration solvers are just variations of this sum, be it the rectangle rule, the trapezoidal rule, Gaussian quadrature and so on. Those methods have some rule through which the integration points x_i and weights w_i are chosen. In Monte Carlo integration the weights are all equal to 1, while the integration points are chosen randomly. If the numbers are drawn from a uniform distribution on $[a, b]$ and the number of samples N is large the sum should yield a result close to the exact integral.

For lower-dimensional integrals the Monte Carlo method is generally inferior to the high quality quadrature methods. The standard deviation of the Monte Carlo method goes like $\sigma \propto 1/\sqrt{N}$ (see section 10.2 in ref. [18]). Quadrature methods provide errors that go like $\sigma \propto h^k \propto N^{-k}$, where h is the separation distance of the integration points and k is a positive integer.

For lower-dimensional integrals it is clear that even the "bad" integration methods like the trapezoidal method is superior to Monte Carlo. But let us consider a hypercube with side L and dimension d. The cube will contain $N = (L/h)^d$ points and therefore the error will scale as $N^{-k/d}$. Since the error of the Monte Carlo method is independent of the number of dimensions it will be more suitable as the number of dimensions increase. It is clear that we can expect Monte Carlo to be preferable when $d > 2k$ for our best quadrature method.

3.3.1 Importance Sampling

When the function we are integrating varies significantly over the integration domain, i.e. there are regions where the function is larger than others, the uniformly distributed integration points may not be optimal. If the function has a narrow peak, we might only wind up with a few integration points sampling that peak. We would like to sample the more important (i.e. the higher valued) regions of the function more. This can be accomplished through importance sampling.

Let $\rho(x)$ be a function on $[a, b]$ which closely mimics $f(x)$, in the sense that $f(x)/\rho(x)$ is nearly flat. Also we require ρ to be normalized

$$\int_a^b \rho(x)\, dx = 1.$$

We then write the integral as

$$\int_a^b f(x)\, dx = \int_a^b \rho(x) \left[\frac{f(x)}{\rho(x)}\right] dx.$$

The function in square brackets is nearly flat, and the $\rho(x)$ in front of the brackets can be included by drawing our random numbers from it instead of the uniform distribution. When we then sample the function in brackets with random points drawn from ρ we are sampling a near-flat function, which will result in a more precise answer (i.e. the standard deviation of the result will be lower).

3.4 The Metropolis Algorithm

The Metropolis algorithm is a so-called Markov chain Monte Carlo method for obtaining a sequence of random samples from a probability distribution for which direct sampling is difficult.

3.4.1 Markov Chains

A Markov chain is, as the name suggests, a chain of objects (numbers or the state of a system for example). For an uncorrelated chain the probability of occurrence of a particular sequence of N objects $X_1, ..., X_N$ is statistically uncorrelated

$$P_N(X_1, ..., X_N) = P_1(X_1) ... P_1(X_N), \tag{3.2}$$

where $P_1(X)$ is the probability of occurrence for object X, and this probability is the same for each step. For example a true random number generator would produce an uncorrelated chain of numbers.

On the other hand in a Markov chain the next step depends on the previous step (but does not depend on any other steps, i.e. the history of the chain is irrelevant). This is described by the transition probability $T(X \to X')$, which is the probability that the object X' will be the next step after X.

The probability for eq. (3.2) would then be

$$P_N(X_1, ..., X_N) = P_1(X_1) T(X_1 \to X_2) T(X_2 \to X_3) ... T(X_{N-1} \to X_N).$$

The transition probabilities must obviously be normalized

$$\sum_{X'} T(X \to X') = 1,$$

so that something always happens (the case where we stay at the same object is also valid, i.e. we can have $X = X'$).

3.4.2 Generating a Markov Chain

We want to be able to generate a Markov chain of system configurations that corresponds to a given distribution. We want the Markov chain to sample our distribution independent of the position in the chain and the initial configuration of the chain.

The Markov chain must therefore be ergodic [18], which means it must fulfill these conditions:

- Every configuration which we want to be included in the ensemble should be accessible from every other configuration within a finite number of steps (this is called connectedness or irreducibility).
- There should be no periodicity. Periodicity would mean that after visiting a particular configuration, it should not be possible to return to the same configuration except after $t = nk$ steps, $n = 1, 2, 3...$, where k is fixed.

The Metropolis Monte Carlo method consists of generating a Markov chain of configurations that correspond to a given distribution. We need to find a transition probability $T(X \to X')$ that leads to our given distribution $\rho(X)$ being sampled.

We introduce the function $\rho(X,t)$, which represents the probability of having configuration X at time/Markov step t. The change in this function from one step to another is governed by two things: going from a configuration X at time t to a configuration X' at $t+1$, which decreases $\rho(X)$, and going from X' at t to X at $t+1$, which increases $\rho(X)$.

This can be summarized in the so-called master equation (see section 10.3 in ref. [18])

$$\rho(X,t+1)-\rho(X,t)=-\sum_{X'}T(X\to X')\,\rho(X,t)+\sum_{X'}T(X'\to X)\,\rho(X',t).$$

We want to find the stationary distribution, where $\rho(X,t+1)=\rho(X,t)$. This gives us

$$\sum_{X'}T(X\to X')\,\rho(X,t)=\sum_{X'}T(X'\to X)\,\rho(X',t).$$

One solution is immediately apparent (we also drop the t dependence of ρ)

$$T(X\to X')\,\rho(X)=T(X'\to X)\,\rho(X').$$

This is called the detailed balance solution. It indicates that the "flow" of states from X to X' is balanced by the opposite flow from X' to X.

We now split the transition probability into two parts

$$T(X\to X')=\omega_{XX'}A_{XX'},$$

where $\omega_{XX'}$ is the trial step probability of our algorithm, i.e. the probability that we will try to move from X to X', and $A_{XX'}$ is the acceptance probability, i.e. the probability that we will accept the move if a move from X to X' is proposed. Since both $\omega_{XX'}$ and $A_{XX'}$ are probabilities, they must both be a number between 0 and 1.

Substituting this form into the detailed balance solution gives us

$$q_{XX'}=\frac{A_{XX'}}{A_{X'X}}=\frac{\omega_{X'X}\rho(X')}{\omega_{XX'}\rho(X)}.$$

Our algorithm then consists of first selecting a new state X' with a probability $\omega_{XX'}$. In the second stage we compare the weights (including the possibly different transition probabilities) of the old state and the new state. We choose $A_{XX'}$ equal to 1 if $\omega_{X'X}\rho(X')>\omega_{XX'}\rho(X)$, else it is chosen equal to $\omega_{X'X}\rho(X')/\omega_{XX'}\rho(X)$. This means that we accept the new state with probability $A_{XX'}=\min(1,q_{XX'})$.

3.5 Variational Monte Carlo

Variational Monte Carlo (VMC) is a method where we parametrize a trial wave function, then calculate the energy of this wave function while varying the parameters to try and find a minimum.

We want to calculate the energy, given by the stationary Schrödinger equation

$$E = \frac{\int \psi^* (R) \hat{H} \psi (R) \, dR}{\int \psi^* (R) \psi (R) \, dR} = \frac{\langle \psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle},$$

where $R = \mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N$ is the set of particle coordinates. We will rewrite this equation as

$$\langle E_L \rangle = \int P (R) \, E_L (R) \, dR,$$

where

$$P (R) = \frac{\psi_T^2 (R)}{\int \psi_T^2 (R) \, dR},$$

is the normalized probability distribution (we won't have to calculate the normalization integral), and

$$E_L = \frac{\hat{H} \psi_T (R)}{\psi_T (R)},$$

is called the local energy. Here ψ_T is the trial wave function with some variational parameters. It is assumed to be real. The mean of a large number of samples of E_L will be our estimate of the ground state energy, and this is the quantity we want to calculate and minimize. We also note that if ψ_T were the exact eigenfunction of the Hamiltonian, E_L would be constant, which would imply a variance of 0 in our calculations. We can therefore also keep an eye on the variance as we vary the parameters, since both the local energy and variance are likely to have minima close to each other.

We now want to use the Metropolis algorithm to find $\langle E_L \rangle$. We solve the integral by using $P (R)$ as the distribution we want to sample, noting that since we always compute the ratios of this distribution in the Metropolis algorithm we won't have to calculate the normalization integral $\int \psi_T^2 (R) \, dR$. As the Markov chain evolves according to this distribution we will calculate the local energy E_L after having proposed moves for all particles.

3.5.1 Importance Sampling

While it is possible to use a simple random selection of a new position for each time step (brute force sampling), that method uses an arbitrary step length regardless of where in space a particle is located, which is not optimal (the step length parameter must be tuned manually). Importance sampling allows us to use a variable step length, and to introduce a bias toward the more interesting areas (i.e. where the wave function, and thus also the probability, is larger). We will select a new position for each particle by the formula ([18] eq. 12.42)

$$\mathbf{r}_{\text{new}} = \mathbf{r}_{\text{old}} + \chi + D\mathbf{F} (R) \, \Delta t. \tag{3.3}$$

Where χ is a random variable with normal distribution, mean equal zero and variance $2D\Delta t$. The diffusion constant is $D = \frac{1}{2}$, which follows from the Schrödinger equation. The quantum force $\mathbf{F}$ introduces the mentioned bias towards areas where the trial wave function is larger, and is given by (see eq. (12.47) in ref. [18])

$$\mathbf{F}(R) = 2\frac{1}{\psi_T(R)}\nabla\psi_T(R). \tag{3.4}$$

The quantum force $\mathbf{F}$ thus pushes the otherwise random selection of the new position in the direction of the gradient of the wave function.

Our new position selection has an associated transition probability given by a Green's function (see eq. (12.44) in ref. [18])

$$\omega_{XX'} = G(R, R', \Delta t) = \exp\left(-\frac{\left(R' - R - D\Delta t\mathbf{F}(R)\right)^2}{4D\Delta t}\right),$$

where R' contains the proposed move $\mathbf{r}_{\text{new}}$. Our acceptance test in the metropolis algorithm is now

$$q_{XX'} = \frac{G(R', R, \Delta t)\cdot\psi_T^2(R')}{G(R, R', \Delta t)\cdot\psi_T^2(R)}. \tag{3.5}$$

3.6 Diffusion Monte Carlo

Diffusion Monte Carlo is more thoroughly introduced in chapter 12 of ref. [18] and chapter 3 of ref. [12], but the outline is given here. The idea behind it is to solve the Schrödinger equation in complex time, making the substitution $t \to i\tau$,

$$-\frac{\partial\psi(R,\tau)}{\partial\tau} = [H - E]\,\psi(R,\tau).$$

The formal solution is given by

$$\psi(R,\tau) = e^{-[H-E]\tau}\psi(R,0),$$

where $G = e^{-[H-E]\tau}$ is the Green's function (basically the propagator discussed in section 2.2.2), and E is a convenient energy shift.

In DMC the wave function is actually represented by a set of walkers (where a walker is a full system in its own right). For the case of bosons the wave function is positive definite everywhere, and can therefore be considered a probability distribution function.

The DMC method involves Monte Carlo integration of the Green's function by every walker. We do the time evolution in small steps, approximating the Green's function as

$$G = e^{-[H-E]\tau} = \prod_{i=i}^{n} e^{[H-E]\Delta\tau},$$

where $\Delta\tau = \tau/n$. If we assume that the starting state can be expanded in the basis of stationary states

$$\psi(R,0) = \sum_\nu C_\nu \phi_\nu(R),$$

we get

$$\psi(R,\tau) = \sum_\nu e^{-[E_\nu - E]\tau} C_\nu \phi_\nu(R),$$

Given enough time this will pick out the ground state contribution (assuming $C_0 \neq 0$). We will in practice adjust the energy shift E so that $E \approx E_0$. The exponential will then cause higher energy states to become negligible over time. If we set E too low all contributions disappear over time, while with E too high we get unrestrained growth.

We can note that the above approach does not need any information about the wave function, as the initial distribution can be virtually anything. While this is a very nice property (we can find the ground state energy without having any idea what the ground state wave function looks like!), it is not very efficient.

Again we introduce the concept of importance sampling. We substitute the wave function $\psi(R,\tau)$ with a new quantity $f(R,\tau) = \psi_T(R)\psi(R,\tau)$, where ψ_T is a trial wave function that we believe to be a good approximation to the ground state wave function (the trial wave function from VMC would be a natural choice). The Green's function that results can be split into a branching part and a diffusion part (see eq. 3.52 and 3.55 in ref. [12]). The diffusion part is the same as it was for importance sampled VMC, namely

$$G(R,R',\Delta\tau) = \exp\left(-\frac{(R' - R - D\Delta\tau\mathbf{F}(R))^2}{4D\Delta\tau}\right),$$

where

$$\mathbf{F}(R) = 2\frac{1}{\psi_T(R)}\nabla\psi_T(R),$$

is the quantum force. The diffusion part is used by each individual walker just like in the VMC case. The new feature then is the branching part. The relevant Green's function is

$$G_B(R,R',\Delta\tau) = \exp\left(-\Delta\tau\left(\frac{E_L(R') + E_L(R)}{2} - E_T\right)\right),$$

and it adjusts the distribution of walkers.

At its core we see that DMC turns out to be very similar to VMC, except that now we must run several VMC simulations in parallel (where one such simulation is called a walker). The branching Green's function is implemented by removing or adding new walkers after each time step. This is handled by the equation (see chapter 12 in ref. [18])

$$\begin{aligned} q &= \exp\left(-\Delta\tau\left(\frac{E_L(R') + E_L(R)}{2} - E_T\right)\right) \\ s &= \text{floor}(q + r), \end{aligned}$$

Algorithm 3.1 Variational Monte Carlo.

```
Generate initial random particle configuration
for 0 to Monte Carlo cycles
  for 0 to number of particles
    Move particle
    Compute wave function and Green's function ratios
    Accept/reject move according to Metropolis prob.
  If thermalisation is done, sample observables
Use sampled values to compute final results
```

where E_L is the local energy as previously defined, r is a uniform random number on $(0, 1)$, and E_T is a trial energy that we adjust so that the number of walkers stays fairly constant. The integer s tells us whether the walker survives. If $s = 0$ the walker is killed, if $s = 1$ the walker lives and if $s > 1$ we make $s - 1$ additional copies of the walker.

A DMC simulation can then be performed by first simulating the desired number of walkers by the VMC algorithm until we think the distribution is good, and then we simply turn on the branching part.

It should be noted that from now on Δt will be used when discussing the time step both for VMC and DMC since much of the discussion will be common between the two. From an implementation point of view there is no difference between Δt and $\Delta\tau$.

Note also that the DMC algorithm is in principle exact, i.e. we are picking out the true ground state energy. The accuracy of the result is only limited by numerical precision, statistical errors and time step errors.

3.7 Summing Up the Algorithms

The VMC and DMC approaches are summed up in algorithms 3.1 and 3.2. We note that the DMC algorithm is simply comprised of several VMC-like "walkers" (parallel instances of systems), except we also have the branching part where some walkers are removed, some are untouched, while others spawn copies.

Algorithm 3.2 Diffusion Monte Carlo.

```
Generate initial random configurations for several walkers
Do VMC algorithm on the walkers until thermalized
for 0 to Monte Carlo cycles
  for 0 to number of walkers
    for 0 to number of particles in walker
      Move particle
      Compute wave function and Green's function ratios
      Accept/reject move according to Metropolis prob.
    Calculate energy of walker
    Calculate s = floor(q + r)
    If s == 0 remove walker
    If s == 1 do nothing
    If s > 1  create s-1 additional copies of walker
  If thermalisation is done, sample observables
Use sampled values to compute final results
```

Chapter 4

The Systems

This section introduces the systems that will be simulated in this thesis. There are two different systems: Atoms and Bose-Einstein condensates (BEC). The atoms simulated are helium, beryllium and neon, while the BEC is modeled after the Anderson *et al.* [21] experiment.

4.1 Atoms

The first system we will look at is the atomic case, where the atoms that will be simulated are helium, beryllium and neon. This subsection describes the physics of these systems and the methods used to simulate them. Note that we will be using the Born-Oppenheimer approximation, meaning that the nucleus is considered to be stationary at the center of our coordinate system. Qualitatively this approximation makes sense since the mass of a nucleon is three orders of magnitude larger than the mass of an electron.

For the atoms we will be working with atomic units, where

$$\hbar = m_e = e = \frac{1}{4\pi\epsilon_0} = 1.$$

This means that all energies calculated in this system must be multiplied by $2E_0 = 27.2$ eV to get the physical value. We will denote the energy unit E_h.

4.1.1 Hamiltonian

The Hamiltonian of the atomic system takes the form

$$\begin{aligned} H &= T + V \\ &= -\frac{1}{2}\sum_{k=1}^{N}\nabla_k^2 - Z\sum_{k=1}^{N}\frac{1}{r_k} + \sum_{i<j}^{N}\frac{1}{r_{ij}}, \end{aligned}$$

where N is the number of electrons (2, 4 and 10 respectively for helium, beryllium and neon), Z is the charge of the nucleus (which is the same as the number of electrons since the atoms are neutrally charged), r_k is the electron-nucleus distance of electron k, and r_{ij} is the distance between electrons i and j. The first sum accounts for the kinetic energy of the electrons, while the second sum is the potential energy of the electrons because of the interaction with the nucleus. Finally the double-sum accounts for the electron-electron interactions.

4.1.2 Wave Function

We will use the hydrogenic orbitals as the building blocks for our trial wave functions

$$\begin{aligned} \phi_{1s}(\mathbf{r}_i) &= e^{-\alpha r_i} \\ \phi_{2s}(\mathbf{r}_i) &= \left(1 - \frac{\alpha r_i}{2}\right) e^{-\alpha r_i/2} \\ \phi_{2p}(\mathbf{r}_i) &= \alpha \mathbf{r}_i e^{-\alpha r_i/2}. \end{aligned}$$

The hydrogenic wave functions can be a good starting point for atomic orbitals. Note that the 1s and 2s orbitals only have spin degeneracy (so there can only be two electrons in each state, one with spin up and the other with spin down) while the 2p orbital has three additional levels of degeneracy based on whether the momentum projection is along the x, y or z axis. These three degeneracies are selected based on which component of $\mathbf{r}_i$ is used in the computation of ϕ_{2p}. For the 2p orbital there are totally 6 possible states. For example $\phi_{2p}(\mathbf{r}_i) = \alpha x e^{-\alpha r_i/2}$ would be two of them (one with spin up and one with spin down), where x is the x-component of $\mathbf{r}_i$.

Our trial wave functions will have two variational parameters, denoted α and β. Here we can view α as the effective charge of the nucleus for the hydrogenic orbitals, which will be slightly less than the actual charge since the other electrons somewhat lessen the effective charge. The β parameter is for the so-called Jastrow factor, which is meant to account for the electron-electron correlation.

Since we are dealing with electrons, which are fermions, our trial wave function will be a Slater determinant (see section 2.4.4) of the form

$$\begin{aligned} \psi_T(R) &= \psi_D \cdot \psi_C \\ &= \mathrm{Det}\left(\phi_1(\mathbf{r}_1), \phi_2(\mathbf{r}_2), ..., \phi_N(\mathbf{r}_N)\right) \cdot J_n, \end{aligned}$$

where $\psi_D = \mathrm{Det}\left(\phi_1(\mathbf{r}_1), \phi_2(\mathbf{r}_2), ..., \phi_N(\mathbf{r}_N)\right)$ is a Slater determinant, and $\psi_C = J_n$ is the Jastrow factor, meant to account for the complicated electron-electron correlations that are not adequately handled by a single Slater determinant. Three possible forms for the Jastrow factor are:

$$
\begin{aligned}
J_1 &= \prod_{i<j}^{N} \exp\left(\frac{a r_{ij}}{1+\beta r_{ij}}\right) \\
J_2 &= \prod_{i<j}^{N} \exp\left(a\left(1+\beta_1 r_{ij}+\beta_2 r_{ij}^2\right)\right) \\
J_3 &= \prod_{i<j}^{N} \exp\left(\frac{a\left(1+\beta_1 r_{ij}+\beta_2 r_{ij}^2\right)}{1+\beta_3 r_{ij}}\right).
\end{aligned}
$$

The factor a in the Jastrow factor is $\frac{1}{2}$ when particle i and j have same spin, and $\frac{1}{4}$ when opposite (see page 147 in ref. [1]). I will only be using J_1 in this thesis.

The goal is to find for which pair of values of α and β the total energy is minimized, where α only appears in the Slater determinant and β only appears in the Jastrow factor. The hope is that for these values the trial wave function will closely mimic the exact solution.

4.2 Bose-Einstein Condensates

In addition to the atoms described above, a Bose-Einstein condensate (BEC) modeled after the experiment by Anderson *et al.* [21] will be simulated using VMC and DMC methods.

4.2.1 Overview

This section provides an overview of the BEC that will be simulated. The paper by Dalfovo [14] gives a thorough overview of these kinds of systems.

The system to be simulated is a gas of rubidium-87 trapped in a potential that can be approximated by a harmonic oscillator potential, as done in the experiment by Anderson *et al.* The atoms are neutrally charged bosons. In Anderson's experiment BEC was observed at a temperature less than 170 nanokelvin and a number density of $2.5 \times 10^{12}\text{cm}^{-3}$.

A key feature of the trapped alkali and atomic hydrogen systems is that they are dilute. The characteristic dimensions of a typical trap for ^{87}Rb is $a_\perp = (\hbar/m\omega_\perp)^{1/2} \approx 10^3$ nm. Here $\omega_\perp$ is the frequency of the harmonic oscillator in the x-y plane (or the general frequency if the trap is not deformed), further discussed in section 4.2.2.

The interaction between ^{87}Rb atoms can be well represented by its s-wave scattering length, a_{Rb}. This scattering length lies in the range $85 < a_{\text{Rb}} < 140\ a_0$ where $a_0 = 0.0529177$ nm is the Bohr radius. The definite value $a_{\text{Rb}} = 100\ a_0$ is usually selected and for calculations the definite ratio of atom size to trap size $a_{\text{Rb}}/a_\perp = 4.33 \times 10^{-3}$ is usually chosen. We use the same values as

others [13, 24, 25] since we are primarily interested in verifying the results of the implementation.

We will be working in harmonic oscillator units where the energy is given in units of $\hbar\omega_\perp$ and the length unit is the characteristic dimension of the trap (in the x-y plane for deformed traps) $a_\perp = (\hbar/m\omega_\perp)^{1/2}$.

4.2.2 Hamiltonian

We will be using both a spherically symmetric trap and deformed (elliptic) trap. For the deformed trap we have one harmonic oscillator frequency in the x-y plane denoted $\omega_\perp$ and one in the z plane denoted ω_z. For the spherically symmetric trap $\omega_\perp = \omega_z = \omega_{\text{ho}}$. The ratio between the frequencies will be denoted $\lambda = \omega_z/\omega_\perp$, and since $\omega_\perp$ disappears in our system of units only λ will enter the equations. Thus $\lambda = 1$ gives us a spherically symmetric trap, $\lambda < 1$ a cigar shaped trap (the potential is expanded in the z-direction) and $\lambda > 1$ a disk shaped trap (the potential is squeezed in the z-direction). We can note that the ratio of characteristic dimensions is $a_\perp/a_z = (\omega_z/\omega_\perp)^{1/2} = \sqrt{\lambda}$.

The spherically symmetric (S) and deformed elliptic (E) traps we will use are given by the external potentials

$$V_{\text{ext}}(\mathbf{r}) = \begin{cases} \frac{1}{2}r^2 & (S) \\ \frac{1}{2}\left(x^2 + y^2 + \lambda^2 z^2\right), & (E) \end{cases} \tag{4.1}$$

and the two-body Hamiltonian of the system is given by

$$H = \sum_i^N \left(-\frac{1}{2}\nabla_i^2 + V_{\text{ext}}\left(\mathbf{r}_i\right)\right) + \sum_{i<j}^N V_{\text{int}}\left(\mathbf{r}_i, \mathbf{r}_j\right),$$

where the first sum of the Hamiltonian covers the kinetic energy of each atom along with the potential energy due to the harmonic oscillator potential V_{ext}, while the second sum accounts for the interaction between the atoms.

The boson-boson interaction is represented by a pairwise, hard core potential

$$V_{\text{int}}\left(\mathbf{r}_i, \mathbf{r}_j\right) = \begin{cases} \infty & r_{ij} \leq a \\ 0 & r_{ij} > a, \end{cases} \tag{4.2}$$

where a is the hard core diameter of the bosons and r_{ij} is the distance between the two bosons. Clearly, V_{int} is zero if the bosons are separated by a distance r_{ij} greater than a but infinite if they attempt to come within a distance $r_{ij} \leq a$. This will simply be implemented by checking that atoms never come too close. When it happens, new random numbers will be drawn and calculations are repeated. This simple hard core interaction has been compared to other potentials and verified to work well with the DMC method by Blume and Greene in ref. [13].

4.2.3 Wave Function

In the non-interacting limit, the ground-state wave-function for the Hamiltonian operator is

$$\psi(R) = \prod_{i=1}^{N} g(\mathbf{r}_i) = \lambda^{1/4}\pi^{-3/4} \exp\left(-\frac{1}{2}\sum_{i=1}^{N}\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right),$$

which we see is quite similar to the ground state of the harmonic oscillator (eq. (2.3) in section 2.3.2). Furthermore the exact solution for a pair of particles at low energy interacting via a hard core potential of diameter a is (see ref. [28])

$$f(r) = \begin{cases} 1 - \frac{a}{r}, & r > a \\ 0, & r \leq a \end{cases}.$$

Our trial wave function is then chosen to be

$$\begin{aligned} \psi_T(R;\alpha) &= G(R)\,F(R) \\ &= \prod_{i=1}^{N} g(\mathbf{r}_i;\alpha) \prod_{i<j}^{N} f(r_{ij}), \end{aligned}$$

where

$$g(\mathbf{r}_i;\alpha) = \exp\left(-\alpha\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right),$$

with α as our only variational parameter. The constant terms are dropped since we always deal with ratios of wave functions.

Chapter 5

OpenCL and GPGPU

This chapter gives an overview of what OpenCL (Open Computing Language) is, and also looks at the basics of GPGPU (General Purpose computing on Graphics Processing Units).

5.1 Introduction

OpenCL is an open and royalty-free standard for general purpose parallel programming on heterogeneous systems. It allows us to write code once, then run it on a range of devices as long as there is an implementation of OpenCL available for the device.

Currently there are three common ways to do GPGPU, namely Nvidia's Compute Unified Device Architecture (CUDA), OpenCL, and DirectCompute.

CUDA has the advantage of being more quickly developed for new hardware advances from Nvidia, and has many advanced features (such as support for writing code for the GPU in C++). It is limited to Nvidia GPUs (though there is a x86 compiler available from the Portland Group [10]).

OpenCL on the other hand has the advantage of being supported on a much broader set of devices, including GPUs from both the major suppliers (AMD and Nvidia), x86 CPUs (with implementations from both AMD and Intel), and IBM's Power and Cell architectures.

DirectCompute is a part of Microsoft's DirectX API (Application Programming Interface) and works on most modern GPUs, but is specific to the Windows operating system.

5.2 OpenCL Basics

OpenCL is a framework consisting of an API and the OpenCL C language. The API is originally written for C, but bindings for C++, Python, C# etc. also exist. The implementation presented in this thesis uses the C++ bindings.

An OpenCL program will consist of two parts: The host code and kernel code. The host code can be written in any language for which there are OpenCL API bindings available. The kernel code is written in OpenCL C. As the name suggests, OpenCL C is based on C99, with some additions and restrictions.

The host code is responsible for coordinating the overall flow of an OpenCL application. It decides which devices to use, what kernels to launch, and is responsible for transfers between host memory, meaning the host computer's RAM, and device memory, meaning the RAM of the device (for instance on-board RAM on a graphics card).

The kernel code on the other hand consists of kernels (that the host can launch) and functions. Kernels and kernel functions are very similar to C functions, with a few additional data types and keywords.

The parallel nature of OpenCL is expressed in the way we launch kernels. We can launch almost any number of kernels, usually several thousand at the same time when running on a GPU. The only thing that separates the kernels are their IDs, which the kernels will typically use to select the right input data from buffers.

An instance of a kernel is called a work-item. The work-items are grouped into one or more work-groups, and there is a memory region called local memory that is shared between work-items within a work-group. Since the present problem does not call for the use of local memory we won't go into more detail. However work-groups are still important as they decide how work-items are executed on the hardware, which will be described in section 5.3.

The index space on which work-items are mapped is called an NDRange in OpenCL parlance. Work-items can be mapped on a one, two or three-dimensional NDRange. The dimensionality is just a matter of convenience (i.e. if we were dealing with a two-dimensional problem like a matrix a two-dimensional NDRange might make sense). Our implementation is merely launching independent simulations in each work-item, and so we simply use a one-dimensional NDRange with the desired number of work-items.

The general flow of an OpenCL application goes something like this:

1. Query for available platforms (implementations of OpenCL) and select one. For instance there could be one platform from AMD supporting x86 CPUs and AMD GPUs, and one Nvidia platform supporting Nvidia GPUs.

2. Query for devices available in selected platform and select one or more.

3. Create an OpenCL context and assign one or more devices to it. Everything below happens within this context.

4. Create OpenCL buffers (memory objects) in the context and upload any necessary data. Buffers primarily live in device memory (for example a GPU's on-board RAM).

5. Create queues for the devices. Each device has a queue, and there is only one device per queue.

6. Add work (kernel launches and memory transfers) to the queues. Buffers serve as input and output for the kernels.

7. Download results from buffers to host RAM.

This was meant as a quick overview of what an OpenCL application does. For a more thorough introduction the OpenCL specification, available for free from the Khronos Group [9], is excellent.

5.3 Basic GPU Architecture

While a thorough description of GPU and CPU architecture is both outside the scope of this paper and my abilities, it is possible to provide a broad overview of the differences between modern x86 CPUs and GPUs, and what limitations we face when working with GPUs.

CPUs are generally composed of several CPU cores, where each core operates independently (it usually shares a level of cache and the memory controller with other cores). Each core is quite complex, with more than 100 million transistors per core[1] (also counting caches). Each core can generally execute only one instruction at a time (though this is blurred somewhat by simultaneous multithreading). SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) allows a CPU core to perform a few floating point operations in parallel. In the case of AVX operating on 64-bit floating point values Intel's Sandy Bridge CPUs can perform as much as four additions and four multiplications at the same time, for a total of 8 double precision floating point operations in parallel.

GPUs are on this level quite different. When drawing analogies to a CPU it is not entirely clear what, if anything, can be compared to a CPU core. Figure 5.1 shows the basic layout of an AMD Cayman GPU. The GPU contains several (in this case 24) compute units, which is the execution resource on which a work-group executes. Each compute unit will usually have its own on-die[2] local memory. In the case of Cayman, each compute unit contains 16 processing elements. At any one point in time one work-item will be executing on one processing element.

Present high-end AMD hardware is generally designed so that over a four-cycle period the processing elements must execute work-items from the same work-group (see section 1.3.1 in ref. [11]). Since there are 16 processing elements per compute unit, this means that we must have at least 64 work-items per work-group to make proper use of the execution hardware. On Nvidia hardware the equivalent number is usually 32, but these numbers can vary between devices (and of course change in the future).

[1] For example a modern AMD Phenom II six-core CPU has about 758 million transistors (including cache, memory controller and so on).

[2] Meaning it resides on the same microchip as the compute unit, the implication being that this is much faster than off-chip resources like RAM.

With 24 compute units and 16 processing elements per compute unit we see that Cayman can process instructions from $24 \times 16 = 384$ work-items at the same time. However several times more are needed to ensure efficient execution. As already mentioned we require at least 64 work-items per work-group, and in addition experience shows that at least four such work-groups should be available per compute unit to achieve reasonable efficiency. Thus the minimum number of work-items we should launch is $4 \times 4 \times 384 = 6144$. This is merely an observation relevant to my code, and can vary significantly for different kernels and devices.

In addition, on GPUs the compute units are usually SIMD (Single Instruction Multiple Data) processors, meaning every work-item in a work-group must in fact be executing the same instruction at the same time. This is not a limitation imposed by OpenCL, as work-items are free to do anything they want independently of the other work-items in the work-group. However on GPUs this will lead to inefficient execution. For example if there is a control flow statement that is taken by only one work-item, all the other work-items in the work-group must effectively wait for it before continuing. In reality all work-items will be executing the instructions associated with the control flow, however the results will be discarded for all but the one that actually took the divergent control flow path (see section 1.3.2 in ref. [11]). This is entirely transparent to the programmer.

The layout presented in figure 5.1 is quite general in modern GPUs, and a similarly simplified diagram of a Nvidia GPU would look very similar. However one of the major differences between AMD and Nvidia GPU architecture is how the processing elements are built. Nvidia has generally used scalar processing elements, meaning the processing elements have been capable of executing one operation at a time. On the other hand AMD has opted for a VLIW (Very Long Instruction Word) design, where each processing element can execute several operations in parallel.

While the VLIW approach can lead to higher theoretical peak performance (a metric where AMD GPUs often score twice as high as similar offerings from Nvidia) it is also more difficult for the compiler to achieve that performance. In addition to the programmer explicitly expressing parallelism through assigning work to parallel work-items, the compiler also wants there to be plenty of independent operations within each work-item so that it can extract ILP (Instruction Level Parallelism). A way for the programmer to make this easier on the compiler is by using vector operations (i.e. using the built-in vector data types of OpenCL like float4).

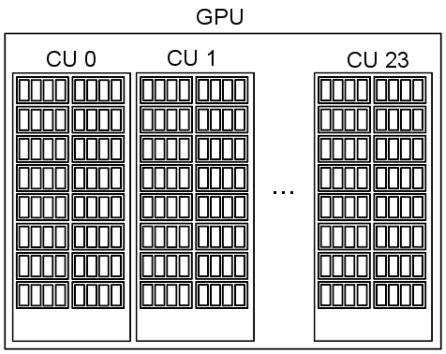


Figure 5.1: Illustration of the general layout of an AMD Cayman GPU.

Since in our particular case we are dealing with particle coordinates in three dimensions this works out very well on VLIW hardware, but in other cases it can be more difficult.

Another general limitation of GPUs is the amount of cache available. The data caches are quite tiny compared to CPUs, and so we are very reliant on being smart when accessing the GPU's RAM. The most straight-forward method of achieving at least somewhat optimized RAM access patterns on GPUs is to have work-items access adjacent memory addresses in parallel. This causes the memory controllers to receive requests for large contiguous memory regions at a time, which is something they are generally very efficient at accessing. The approach taken to achieve this is further discussed in sections 6.6.4 and 7.5.4.

Furthermore there are also quite limited amounts of register space available on GPUs. Registers are mostly used for private variables (i.e. variables private to a single work-items, like an integer declared as *int myint = 0*).

Each processing element has a certain amount of registers. For example the AMD Cayman GPU has 16384 registers per compute unit (see section 4.11.2 in ref. [11]), each register being able to hold 128 bits of data. This is a rather impressive total of 6 MiB (where $1\,\text{MiB} = 1024^2$ byte) of extremely fast[3] register space on the GPU, however recall that we generally have four work-items per processing element in the minimum sized work-group, and in addition we usually need several such work-groups taking turns executing to hide memory latencies. With the above mentioned sweet-spot of 6144 work-items that leaves about 1 KiB ($1\,\text{KiB} = 1024$ byte) of register space per work-item.

This means that registers are a scarce resource, and if we run out of registers performance can really take a nose-dive as the registers will have to be swapped to some other memory region (usually RAM).

As a consequence of the limited register space, we usually want kernels to be as small and use as few and small private variables as possible. This introduces an unfortunate conflict between AMD and Nvidia GPUs. Since AMD GPUs need ILP to perform well, AMD would like us to use vectors in our kernel code (since it is fairly easy to extract ILP from vector operations). On the other hand Nvidia hardware does not generally benefit from such vectorization.

Since AMD wants us to use vectors they generally have more register space, so while our vectorized kernel might perform well on AMD hardware it could potentially run out of registers on Nvidia hardware. This can make writing optimal code for both AMD and Nvidia hardware difficult, however in our case the algorithm naturally requires vectors, and so there is not much to be done for it.

[3] Total aggregate register read bandwidth is over 16 TB/s for Cayman, see table D.1 in ref. [11].

Chapter 6

The Implementation

Here particulars of the implementation are discussed. We look at the implementation of the RANLUX pseudo-random number generator, results needed to implement the systems such as the closed-form derivatives are presented, and the overall implementation is described.

6.1 Random Number Generation

Monte Carlo algorithms rely on good numbers, the numbers being either random, pseudo-random or have a distribution that we know is suitable (quasi-random). In this thesis pseudo-random numbers will be used. As there is not an abundance of pseudo-random number generators (PRNGs) available for OpenCL, especially if a high-quality and well tested generator is desired, I decided to implement the well-known and high-quality generator RANLUX in OpenCL.

6.1.1 OpenCL RANLUX

Overview of RANLUX

RANLUX [22] is a high quality PRNG proposed by Martin Lüscher in 1994. RANLUX is based on the RCARRY [15] algorithm, which has a very long period ($\sim 10^{171}$).

While RCARRY has been found to fail some tests, Lüscher's idea was to discard some values generated by RCARRY. This is described by the p-value. After 24 values have been generated, $p - 24$ values are thrown away. Lüscher showed that the RCARRY algorithm is related to a classical dynamic system that is chaotic. The statistical correlations between generated variables are exponentially decreasing when p is increased. This allows us to select a tradeoff between speed and quality.

The selection of p-values is conveniently handled through a luxury[1] setting of the generator. In the original Fortran 77 implementation by F. James [16] there are five luxury settings, from 0 through 4. The corresponding p-values are 24, 48, 97, 223 and 389. At luxury setting 4 all 24 bits of the mantissa of a 32-bit floating point number are completely chaotic according to Lüscher [22]. The default value for a decent tradeoff between speed and quality is usually luxury setting 3.

The RANLUX generator uses 24 32-bit floating point seed values. In this implementation (as in the implementation of F. James in ref. [16]) the seed arrays are initialized using a simple multiplicative congruential generator, which is initialized by a single integer (for example based on the computer time). This means that we have $\sim 2^{31}$ sequences available, where each sequence on average can generate $\sim 10^{160}$ random numbers before overlapping with any other sequence. This can be considered adequate for parallel applications, as we will never come close to exhausting the sequences on current computer hardware.

There are several reasons why RANLUX was chosen as the PRNG for this thesis:

- The algorithm is relatively simple to implement, and can be quite efficient on GPUs.
- The algorithm uses floating point calculation instead of integer calculation. This is often faster on GPUs.
- The period is sufficiently large that even huge clusters will not be able to exhaust it.
- Being able to easily select a quality/speed tradeoff is an interesting (and rare) property in a PRNG.
- It is perhaps the oldest high-quality pseudo-random number generator still in use, which taken together with the fact that it has been extensively used in other Monte Carlo projects is a good indication of its quality.

Implementation

The implementation is comprised of a C/C++ initialization function called ranluxcl_initialization(), which is found in the ranluxcl.h header file, while the rest of the generator is implemented in OpenCL C kernel code, and is found in the ranluxcl.cl file.

The host code initialization function is defined as:

[1] The luxury setting is simply the quality setting of the generator. Since RANLUX was and still is somewhat slower at high luxury settings than other generators (but also possibly of better quality) it's numbers are "luxurious", and you must pay for them in computational time. The term was introduced by F. James [16]. In his words:

> On typical platforms, p = 389 runs between five and ten times more slowly than p = 24. For many applications, this time is still negligible; in such cases, the user should not deny himself the luxury of demonstrably good random numbers.

```
cl_float4 *ranluxcl_intialization(
    cl_int lux, cl_int ins, size_t numWorkitems,
    size_t maxWorkitems, cl_int *nskip,
    size_t *RANLUXCLTabSize)
```

where:

lux
Is an integer setting the luxury value for the generator. Can be 0-4 (where 4 is slowest but produces the best numbers), or if lux $\geq$ 24 it directly sets the p-value (lux must then be divisible by 4).

ins
Is an integer seed (ins ≥ 0) used to initialize the generator.

numWorkitems
The number of work-items, meaning parallel generators to initialize.

maxWorkitems
If the generator will be simultaneously used in parallel on different devices (i.e. we will be calling ranluxcl_initialization() more than once and want all sequences to be different) maxWorkitems should reflect the highest value of numWorkitems that will be used in any call to ranluxcl_initialization(). This sets an offset to the actual seed used to initialize the generator that ensures we never have overlapping sequences.

nskip
Returns the value for $p - 24$. Its only use is that it can (optionally) be used to define the macro NSKIP in the OpenCL C kernel code with the given value, which can speed up the implementation somewhat when $p - 24$ is divisible by 24 (like it is for luxury value 0 and 1). This is entirely optional.

RANLUXCLTabSize
Returns the number of bytes allocated for the returned cl_float4 pointer. This should be used as the size argument when creating the associated OpenCL buffer, and when transferring the state array to said buffer.

The initialization function returns a pointer to a cl_float4 array containing the state arrays of all work-items. It should be transferred to the OpenCL device for use by the kernel functions. The kernel functions meant to be called by the user are:

void ranluxcl_download_seed(ranluxcl_state_t *, global float4 *)
Should be called before any other ranluxcl function. Accepts the (uninitialized) ranluxcl state variable and downloads the state data from the provided float4 array.

void ranluxcl_download_seed(ranluxcl_state_t *, global float4 *)
Should be called before any other ranluxcl function. Accepts the (uninitialized) ranluxcl state variable and downloads the state data from the provided float4 array.

void ranluxcl_upload_seed(ranluxcl_state_t *, global float4 *)
Should be called after all needed numbers have been generated. Uploads the state data back into global memory.

void ranluxcl warmup(ranluxcl_state_t *)
Generates (without returning them) the number of values necessary to ensure complete decorrelation. It can be a good idea for each work-item to call this function once after ranluxcl_initialization has been used to generate the state array in host code, to ensure there are no correlations (the default initialization function leaves the generator in an initially correlated state). If this function is not called then the first few calls to ranluxcl() will generate correlated values, however those correlations will quickly disappear. This function is just useful if we want to be sure the parallel streams are uncorrelated from the very beginning.

float4 ranluxcl(ranluxcl_state_t *)
Returns a float4 where each component is a pseudo-random number uniformly distributed on $(0, 1)$, end points not included.

void ranluxcl_synchronize(ranluxcl_state_t *)
Sets the generator to what we can identify as the "beginning" of the algorithm. Useful if different work-items may have called ranluxcl() a different number of times, as the parallel generators may not execute efficiently on SIMD hardware anymore. This function ensures that we are again SIMD-friendly.

The initialization function is basically the same as the one used by the original Fortran 77 implementation by F. James [16]. This has the advantage that the sequences of numbers generated by this OpenCL implementation should be the same as those generated by the original Fortran 77 program. I have developed a C++ application called ranluxcltest that can check that the correct sequences have indeed been generated, and also measures performance. This program (which also includes the generator itself) is available from ref. [7].

After it has been verified that the implementation is generating correct values, the initialization function could be changed if desired. There is nothing "magic" about the way the generator is initialized. We just need to provide 24 initial values per instance/work-item (and a carry bit), where there are only two

possible bad initializations described in Lüscher's paper [22]. For example we could use truly random numbers to initialize the generator, or /dev/random.

As mentioned the implementation is based on the original Fortran 77 implementation by F. James [16]. The OpenCL implementation developed for this thesis uses the same algorithm as the original code by F. James, and generates the same sequences (for the same p-values). However some changes have been made, which are outlined below.

To achieve maximum performance on a GPU it is advantageous to store all state data for the generator in registers. This is accomplished by storing the 24 element seeds array in six float4 vectors. If the seeds array was simply declared as a private array it would likely not be placed in registers, but would have to be emulated in Random Access Memory (RAM). The entire state of the generator is stored in a struct, typedefed as ranluxcl_state_t.

Since the seeds array is no longer stored in an array, the algorithm must be unrolled so that all indexing into the seeds is explicit. A natural consequence of this is that it is convenient to always throw away some multiple of 24 values. This is the approach taken in [30], termed planar RANLUX by the author. However, there are indications [20] that choosing the number of values to discard equal to a multiple of the seeds array length may introduce "resonances", causing slightly stronger correlations.

Therefore a slightly more involved approach is taken, where the indexing is still done explicitly, but the number of values to discard can be any multiple of four instead of 24. This approach causes a slight loss of performance (~ 10 %) compared to the simpler 24 discarding approach, but this is deemed acceptable since it opens up a greater choice of p-values.

The luxury values are then slightly redefined from those proposed by F. James. For luxury values 0 through 4 the p-values are now 24, 48, 100, 224 and 404 respectively (compared to the original 24, 48, 97, 223 and 389). Notice that for luxury values 0 and 1 they are the same, and $p = 404$ is the same value chosen by Lüscher for his v3 version of RANLUX [5] (which is the same algorithm, just a new implementation).

This means that luxury levels 1, 2 and 4 are "official" ones that should be completely safe (or as safe as a PRNG can be at least), while all that can be said for luxury values 2 and 3 is that there is no reason to suspect that they should be bad choices.

6.1.2 Normally Distributed Numbers

RANLUX generates uniformly distributed numbers on the open interval $(0, 1)$. However we will also require normally distributed numbers centered at zero with a standard deviation of one. To generate normally distributed numbers the Box-Muller transform [17] is used.

There are two forms of the Box-Muller transformation, the basic form and polar form. The main difference from an implementation perspective is that the polar form is faster, but has to throw away some of the uniformly distributed input numbers. This is not ideal on SIMD machines like GPUs, since it causes

the execution to fall out of sync. Especially considering the way RANLUX is implemented it is desirable that each OpenCL work-item stays in sync, else we may have inefficient execution.

Therefore the basic form of the Box-Muller transform is used. Given two uniformly distributed numbers U_1 and U_2 on $(0, 1)$ the basic form is given by

$$\begin{aligned} R &= \sqrt{-2\ln(U_1)} \\ \phi &= 2\pi U_2 \\ X_1 &= R \cdot \cos\phi \\ X_2 &= R \cdot \sin\phi, \end{aligned}$$

where X_1 and X_2 are two normally distributed numbers with mean zero and unit variance.

6.2 Common Considerations

6.2.1 Green's Function Ratio

The Green's function ratio (as discussed in section 3.5.1) is needed along with the wave function ratio when deciding whether to accept a proposed move of a particle. Recall that the Green's function ratio is given by

$$\begin{aligned} &\frac{G(R', R, \Delta t)}{G(R, R', \Delta t)} \\ &= \exp\left(-\frac{(R - R' - D\Delta t F(R'))^2}{4D\Delta t} + \frac{(R' - R - D\Delta t F(R))^2}{4D\Delta t}\right) \\ &= \exp\left(\frac{1}{4D\Delta t}\left((R' - R) - D\Delta t F(R)\right)^2 - \left((R - R') - D\Delta t F(R')\right)\right) \\ &= \exp\left(\frac{1}{2}(R - R')(F(R) + F(R')) + \frac{D\Delta t}{4}\left(F(R)^2 - F(R')^2\right)\right) \\ &= \exp\left(\frac{1}{2}(F(R) + F(R')) \cdot \left(\frac{D\Delta t}{2}(F(R) - F(R')) + R - R'\right)\right). \end{aligned}$$

In our algorithm we only move one particle at a time, however since the quantum force $F(R)$ changes for every particle even when only one of them is moved this means that we must calculate the Green's function for all N particles, and thus also calculate the quantum force for every particle every time we propose a single particle move.

As we will see in section 6.5 the derivative $\frac{\nabla_i \psi_C}{\psi_C}$ needed for computing the quantum force has time complexity $\mathrm{O}(N)$, we do it N times every time we propose a move (once for the quantum force of each particle) and we propose moves N times per Monte Carlo cycle. Thus it seems the Green's function ratio (because of the necessary quantum force update) has time complexity $\mathrm{O}(N^3)$ for each Monte Carlo cycle. However as will be discussed in section 6.4.2 for the boson case, it is possible to reduce the order of this calculation.

6.3 Fermion Considerations

6.3.1 Splitting the Slater Determinant

For the atomic (fermionic) systems we need to update the Slater determinant every time we move a particle.

Following [23], chapter 16, when we compute the expectation value of a spin-independent quantum mechanical operator (like our Hamiltonian) we are free to replace the total antisymmetric wave function with one where we have permuted the arguments such that we first have the $N_\uparrow$ spin-up arguments followed by the remaining $N_\downarrow$ spin-down arguments. This means we can write the total Slater determinant as the product of two smaller determinants, one for the spin-up particles and one for the spin-down particles. Our Slater determinant can then be written

$$\psi_D \propto \psi_{D\uparrow}\psi_{D\downarrow},$$

where

$$\psi_{D\uparrow} = \begin{vmatrix} \phi_1(\mathbf{x}_1) & \phi_1(\mathbf{x}_2) & \cdots & \phi_1(\mathbf{x}_{N/2}) \\ \phi_2(\mathbf{x}_1) & \phi_2(\mathbf{x}_2) & \cdots & \phi_2(\mathbf{x}_{N/2}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{N/2}(\mathbf{x}_1) & \phi_{N/2}(\mathbf{x}_2) & \cdots & \phi_{N/2}(\mathbf{x}_{N/2}) \end{vmatrix},$$

this time without the normalization factor since we will always be interested in ratios of wave functions anyway, and the normalization factors cancel. We also have the similar determinant $\psi_{D\downarrow}$ for the remaining particles, where quantum and particle numbers run from $N/2+1$ to N. Therefore where we previously wrote our trial wave function as

$$\psi_T(R) = \psi_D \cdot \psi_C,$$

we can now write it as

$$\psi_T(R) = \psi_{D\uparrow}\psi_{D\downarrow} \cdot \psi_C,$$

where ψ_C is still one of the Jastrow factors.

It turns out (as we will see later) that we only require the inverse Slater determinant when we do our computations. Therefore we initially generate the Slater determinant matrix, then invert it, and after that we will only keep track of the inverse matrix, which can be done like this ([23] eq. 16.18)[2]

$$D_{kj}^{-1}(\mathbf{x}^{\text{new}}) = \begin{cases} D_{kj}^{-1}(\mathbf{x}^{\text{old}}) - \frac{D_{ki}^{-1}(\mathbf{x}^{\text{old}})}{R_{SD}} \sum_{l=1}^{N} D_{il}^{-1}(\mathbf{x}^{\text{new}}) D_{lj}^{-1}(\mathbf{x}^{\text{old}}) & if\ j \neq i \\ \frac{D_{ki}^{-1}(\mathbf{x}^{\text{old}})}{R_{SD}} & if\ j = i \end{cases},$$

where R_{SD} is the Slater determinant ratio

$$R_{SD} = \frac{\psi_D^{\text{new}}}{\psi_D^{\text{old}}},$$

[2]For the case of $j = i$ the sum is removed because of equation 16.8 in [23], which states that $\sum_k^N D_{ik} D_{kj}^{-1} = \delta_{ij}$

and $D_{ij} = \phi_j(\mathbf{x}_i)$ is entry (i,j) of the Slater determinant, and D_{ij}^{-1} is entry (i,j) in the inverted Slater determinant matrix.

6.3.2 Computing the Wave Function Ratio

In the Metropolis algorithm, we need to calculate the ratio

$$\frac{\psi_T^{\text{new}}}{\psi_T^{\text{old}}} = \frac{\psi_{D\uparrow}^{\text{new}}}{\psi_{D\uparrow}^{\text{old}}} \frac{\psi_{D\downarrow}^{\text{new}}}{\psi_{D\downarrow}^{\text{old}}} \frac{\psi_C^{\text{new}}}{\psi_C^{\text{old}}}.$$

Since evaluating a determinant every time we calculate this ratio is computationally expensive, we would very much like to avoid having to do this. It turns out (see page 468 in ref. [23]) that this can be done very efficiently. If we have moved particle i, and all other particles are in the same location, then the ratio of Slater determinants is given by

$$R_{SD} = \frac{\psi_D^{\text{new}}}{\psi_D^{\text{old}}} = \frac{\psi_{D\uparrow}^{\text{new}}}{\psi_{D\uparrow}^{\text{old}}} \frac{\psi_{D\downarrow}^{\text{new}}}{\psi_{D\downarrow}^{\text{old}}} = \sum_{j=1}^{N} \phi_j\left(x_i^{\text{new}}\right) D_{ij}^{-1}\left(x^{\text{old}}\right).$$

As we saw above we also need the ratio $\frac{\psi_C^{\text{new}}}{\psi_C^{\text{old}}}$ when a single particle i has been moved. This is given by (see page 472 in ref. [23])

$$R_C = \frac{\psi_C^{\text{new}}}{\psi_C^{\text{old}}} = \frac{e^{U_{\text{new}}}}{e^{U_{\text{old}}}} = e^{\Delta U},$$

where

$$\Delta U = \sum_{l=1}^{i-1} \left(f_{li}^{\text{new}} - f_{li}^{\text{old}}\right) + \sum_{l=i+1}^{N} \left(f_{il}^{\text{new}} - f_{il}^{\text{old}}\right),$$

where f_{ik} is the exponent argument in the Jastrow factor, for example for J_1 we have $f_{ik} = \frac{ar_{ik}}{1+\beta r_{ik}}$.

6.4 Boson Considerations

6.4.1 Computing the Wave Function Ratio

Like we did for fermions we need the wave function ratio

$$\frac{\psi_T^{\text{new}}}{\psi_T^{\text{old}}} = \frac{\psi_S^{\text{new}}}{\psi_S^{\text{old}}} \frac{\psi_C^{\text{new}}}{\psi_C^{\text{old}}},$$

where ψ_S is the single particle wave function

$$\psi_S = G(R) = \prod_{i=1}^{N} g\left(\vec{r}_i; \alpha\right),$$

where

$$g\left(\mathbf{r}_i;\alpha\right)=\exp\left(-\alpha\left(x_i^2+y_i^2+\lambda z_i^2\right)\right),$$

and ψ_C is the correlation function

$$\psi_C=F\left(R\right)=\prod_{i<j}^{N}f\left(r_{ij}\right),$$

where

$$f\left(r\right)=\begin{cases}1-\frac{a}{r}, & r>a\\ 0, & r\leq a\end{cases}.$$

The ratio of single particle functions is quite simple, since we don't have a Slater determinant to worry about as we did with fermions it is simply

$$\frac{\psi_S^{\text{new}}}{\psi_S^{\text{old}}}=\frac{G\left(R'\right)}{G\left(R\right)}=\frac{g\left(\mathbf{r}_{i,\text{new}};\alpha\right)}{g\left(\mathbf{r}_{i,\text{old}};\alpha\right)},\tag{6.1}$$

where i denotes the moved particle.

The ratio of the correlation functions requires more consideration. We only want to include those parts of $F\left(R\right)$ that have changed after particle i has been moved. Denoting this quantity F_r, it will clearly be

$$F_r\left(R'\right)=\prod_{j=0}^{i-1}f\left(r'_{ij}\right)\times\prod_{j=i+1}^{N}f\left(r'_{ij}\right),$$

where the prime denotes that the moved particle's new position is contained. Thus the entire wave function ratio can be calculated as

$$\begin{aligned}\frac{\psi_T^{\text{new}}}{\psi_T^{\text{old}}} &= \exp\left(-\alpha\left(x_{i,\text{new}}^2+y_{i,\text{new}}^2+\lambda z_{i,\text{new}}^2\right)+\alpha\left(x_{i,\text{old}}^2+y_{i,\text{old}}^2+\lambda z_{i,\text{old}}^2\right)\right)\\ &\quad\times\left(\prod_{j=0}^{i-1}\frac{f\left(r'_{ij}\right)}{f\left(r_{ij}\right)}\times\prod_{j=i+1}^{N}\frac{f\left(r'_{ij}\right)}{f\left(r_{ij}\right)}\right).\end{aligned}\tag{6.2}$$

6.4.2 Optimizing the Quantum Force Update

In section 6.2.1 the formula for computing the Green's function ratio was given. Its computation requires that we update the quantum force of each particle after every proposed move. By the most direct approach this is has time complexity $\mathrm{O}\left(N^3\right)$ per Monte Carlo cycle, however we can reduce this.

The quantum force is computed as (derivatives are presented in section 6.5)

$$\begin{aligned}\mathbf{F}(r_i) &= 2\frac{\nabla_i\psi_T}{\psi_T} \\ &= 2\left(\frac{\nabla_i\psi_S}{\psi_S}+\frac{\nabla_i\psi_C}{\psi_C}\right) \\ &= 2\left(-2\alpha\left(x_i\hat{\mathbf{x}}+y_i\hat{\mathbf{y}}+\lambda z_i\hat{\mathbf{z}}\right)+\sum_{j\neq i}^{N}\frac{a}{r_{ij}\left(r_{ij}-a\right)}\hat{\mathbf{r}}_{ij}\right).\end{aligned}$$

When only a single particle has been moved from the previous quantum force calculation, it is clear that not all of these computations have to be repeated. Instead we subtract the terms that depend on the moved particle (using the old particle coordinates), then add the same terms computed using the new particle coordinates. Clearly we will only have to do the full sum over j above for the quantum force of the moved particle. For all the other particles we only have to compute the one term that accounts for the interaction with the moved particle. Thus the quantum force update is now approximately of time complexity $\mathrm{O}\left(N^2\right)$ instead of $\mathrm{O}\left(N^3\right)$.

The procedure described above where we subtract and then add to the quantum force could conceivably lead to round-off errors after a large amount of cycles, however the quantum force is recomputed from scratch at the beginning of each Monte Carlo cycle, making this a non-issue.

6.5 Derivatives

Calculating the energy and quantum force for use in importance sampling means we need to be able to calculate some derivatives. For both the atomic and Bose-Einstein condensate cases we have both a single particle wave function ψ_S (which is the Slater determinant ψ_D for fermions) and a correlation function ψ_C, where the total wave function is $\psi=\psi_S\psi_C$. We will for example have need of both

$$\begin{aligned}\frac{\nabla_i\psi}{\psi} &= \frac{\nabla_i\left(\psi_S\psi_C\right)}{\psi_S\psi_C} \\ &= \frac{\psi_C\nabla_i\psi_S+\psi_S\nabla_i\psi_C}{\psi_S\psi_C} \\ &= \frac{\nabla_i\psi_S}{\psi_S}+\frac{\nabla_i\psi_C}{\psi_C},\end{aligned}$$

and

$$\begin{aligned}\frac{\nabla_i^2\psi}{\psi} &= \frac{\nabla_i^2\left(\psi_S\psi_C\right)}{\psi_S\psi_C} \\ &= \frac{\nabla_i\left(\nabla_i\left(\psi_S\psi_C\right)\right)}{\psi_S\psi_C} \\ &= \frac{\nabla_i\left(\psi_C\nabla_i\psi_S+\psi_S\nabla_i\psi_C\right)}{\psi_S\psi_C} \\ &= \frac{\nabla_i^2\psi_S}{\psi_S}+\frac{\nabla_i^2\psi_C}{\psi_C}+2\frac{\nabla_i\psi_S}{\psi_S}\cdot\frac{\nabla_i\psi_C}{\psi_C},\end{aligned}$$

where we are only interested in the derivative with regards to particle i.

This means we need $\frac{\nabla_i\psi_S}{\psi_S}$, $\frac{\nabla_i\psi_C}{\psi_C}$, $\frac{\nabla_i^2\psi_S}{\psi_S}$, and $\frac{\nabla_i^2\psi_C}{\psi_C}$. While it is possible to implement these numerically, that would mean several extra calls to transcendental functions since our trial functions use both exponential functions and distances (implying square roots), which would severely limit the performance of our program. Therefore it is desirable to have analytic solutions to these derivatives.

6.5.1 Fermions

Here the closed form derivatives for the fermionic (atomic) systems will be presented.

Slater Determinant

As has been described, we have chosen to split the Slater determinant so it is now written as $\psi_D = \psi_{D\uparrow}\psi_{D\downarrow}$. The derivatives mentioned above will now take the form

$$\begin{aligned}\frac{\nabla_i\psi}{\psi} &= \frac{\nabla_i\psi_S}{\psi_S}+\frac{\nabla_i\psi_C}{\psi_C} \\ &= \frac{\nabla_i\left(\psi_{D\uparrow}\psi_{D\downarrow}\right)}{\psi_{D\uparrow}\psi_{D\downarrow}}+\frac{\nabla_i\psi_C}{\psi_C} \\ &= \frac{\nabla_i\psi_{D\uparrow}}{\psi_{D\uparrow}}+\frac{\nabla_i\psi_{D\downarrow}}{\psi_{D\downarrow}}+\frac{\nabla_i\psi_C}{\psi_C},\end{aligned}$$

and

$$\begin{aligned}\frac{\nabla_i^2\psi}{\psi} &= \frac{\nabla_i^2\psi_S}{\psi_S}+\frac{\nabla_i^2\psi_C}{\psi_C}+2\frac{\nabla_i\psi_S}{\psi_S}\cdot\frac{\nabla_i\psi_C}{\psi_C} \\ &= \frac{\nabla_i^2\left(\psi_{D\uparrow}\psi_{D\downarrow}\right)}{\psi_{D\uparrow}\psi_{D\downarrow}}+\frac{\nabla_i^2\psi_C}{\psi_C}+2\frac{\nabla_i\left(\psi_{D\uparrow}\psi_{D\downarrow}\right)}{\psi_{D\uparrow}\psi_{D\downarrow}}\cdot\frac{\nabla_i\psi_C}{\psi_C} \\ &= \frac{\nabla_i^2\psi_{D\uparrow}}{\psi_{D\uparrow}}+\frac{\nabla_i^2\psi_{D\downarrow}}{\psi_{D\downarrow}}+\frac{\nabla_i^2\psi_C}{\psi_C}+2\left(\frac{\nabla_i\psi_{D\uparrow}}{\psi_{D\uparrow}}+\frac{\nabla_i\psi_{D\downarrow}}{\psi_{D\downarrow}}\right)\frac{\nabla_i\psi_C}{\psi_C}.\end{aligned}$$

The relevant derivatives are ([23] page 470-471)

$$\begin{aligned}\frac{\nabla_i \psi_D}{\psi_D} &= \sum_{j=1}^{N} \nabla_i \phi_j(\mathbf{x}_i) D_{ji}^{-1}(\mathbf{x}) \\ \frac{\nabla_i^2 \psi_D}{\psi_D} &= \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{x}_i) D_{ji}^{-1}(\mathbf{x}).\end{aligned}$$

We see that we will need the inverse Slater determinant matrix and the first and second derivatives of the basis functions. These will be presented in the following sections.

Basis Functions

Here the analytic derivatives of the basis functions for the atomic systems are presented, meaning the hydrogenic basis functions. The functions are

$$\begin{aligned}\phi_{1s}(\mathbf{r}_i) &= e^{-\alpha r_i} \\ \phi_{2s}(\mathbf{r}_i) &= \left(1 - \frac{\alpha r_i}{2}\right) e^{-\alpha r_i/2} \\ \phi_{2p}(\mathbf{r}_i) &= \alpha \vec{r}_i e^{-\alpha r_i/2}.\end{aligned}$$

In the program these functions are denoted ϕ_j, where $j = 0$ corresponds to 1s, $j = 1$ to $2s$ and $j = 2,\ 3$ and 4 are the three versions of the $2p$ orbital, where we use the $x,\ y$ and z component of $\vec{r}_i$ respectively.

As we have seen we will need $\nabla\phi$ and $\nabla^2\phi$ for these functions. The definitions for these operators are

$$\begin{aligned}\nabla f &= \frac{\partial f}{\partial x}\hat{\mathbf{x}} + \frac{\partial f}{\partial y}\hat{\mathbf{y}} + \frac{\partial f}{\partial z}\hat{\mathbf{z}} \\ \nabla^2 f &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2},\end{aligned}$$

or in spherical coordinates

$$\begin{aligned}\nabla f &= \frac{\partial f}{\partial r}\hat{\mathbf{r}} + \text{angular part} \\ \nabla^2 f &= \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2 \frac{\partial f}{\partial r}\right) + \text{angular part}\end{aligned}$$

where $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ are unit vectors along the x, y and z axes, and $\hat{\mathbf{r}}$ is the unit

vector along $\mathbf{r}$. The results are

$$\begin{aligned}
\nabla\phi_{1s}(\mathbf{r}_i) &= -\hat{\mathbf{r}}_i\alpha e^{-\alpha r_i} \\
\nabla\phi_{2s}(\mathbf{r}_i) &= -\hat{\mathbf{r}}_i\frac{1}{4}\alpha(4-r_i\alpha)e^{-\alpha r_i/2} \\
\nabla\phi_{2p_x}(\mathbf{r}_i) &= \alpha e^{-\alpha r_i/2}\hat{\mathbf{x}} - \frac{\alpha^2 r_{i,x}e^{-\alpha r_i/2}}{2r_i}\mathbf{r}_i \\
\nabla\phi_{2p_y}(\mathbf{r}_i) &= \alpha e^{-\alpha r_i/2}\hat{\mathbf{y}} - \frac{\alpha^2 r_{i,y}e^{-\alpha r_i/2}}{2r_i}\mathbf{r}_i \\
\nabla\phi_{2p_z}(\mathbf{r}_i) &= \alpha e^{-\alpha r_i/2}\hat{\mathbf{z}} - \frac{\alpha^2 r_{i,z}e^{-\alpha r_i/2}}{2r_i}\mathbf{r}_i \\
\nabla^2\phi_{1s}(\mathbf{r}_i) &= \frac{e^{-\alpha r_i}\alpha(r_i\alpha-2)}{r_i} \\
\nabla^2\phi_{2s}(\mathbf{r}_i) &= -\frac{e^{-\alpha r_i/2}\alpha\left(16-10r_i\alpha+r_i^2\alpha^2\right)}{8r_i} \\
\nabla^2\phi_{2p_x}(\mathbf{r}_i) &= r_{i,x}\frac{e^{-\alpha r_i/2}\alpha^2(r_i\alpha-8)}{4r_i} \\
\nabla^2\phi_{2p_x}(\mathbf{r}_i) &= r_{i,y}\frac{e^{-\alpha r_i/2}\alpha^2(r_i\alpha-8)}{4r_i} \\
\nabla^2\phi_{2p_x}(\mathbf{r}_i) &= r_{i,z}\frac{e^{-\alpha r_i/2}\alpha^2(r_i\alpha-8)}{4r_i},
\end{aligned}$$

where $r_{i,x}$ is the x-component of $\mathbf{r}_i$ (and similarly for y and z).

Jastrow Factors

Three different Jastrow factors for the atomic systems have been presented. They are

$$\begin{aligned}
J_1 &= \prod_{i<j}^{N}\exp\left(\frac{ar_{ij}}{1+\beta r_{ij}}\right) \\
J_2 &= \prod_{i<j}^{N}\exp\left(a\left(1+\beta_1 r_{ij}+\beta_2 r_{ij}^2\right)\right) \\
J_3 &= \prod_{i<j}^{N}\exp\left(\frac{a\left(1+\beta_1 r_{ij}+\beta_2 r_{ij}^2\right)}{1+\beta_3 r_{ij}}\right).
\end{aligned}$$

We need expressions for

$$\frac{\nabla_i\psi_C}{\psi_C},$$

and

$$\frac{\nabla_i^2\psi_C}{\psi_C}.$$

From ref. [23] page 473-476 and ref. [1] page 145 we get[3]

$$\begin{aligned}\frac{\nabla_i \psi_C}{\psi_C} &= \sum_{j\neq i}^{N} \hat{\mathbf{r}}_{ji} \frac{\partial f_{ji}}{\partial r_{ji}} \\ \frac{\nabla_i^2 \psi_C}{\psi_C} &= \sum_{l,m\neq i}^{N} \hat{\mathbf{r}}_{li} \cdot \hat{\mathbf{r}}_{mi} \frac{\partial f_{li}}{\partial r_{li}} \frac{\partial f_{mi}}{\partial r_{mi}} + \sum_{l\neq i}^{N} \left(\left(\frac{d-1}{r_{li}} \right) \frac{\partial f_{li}}{\partial r_{li}} + \frac{\partial^2 f_{li}}{\partial r_{li}^2} \right),\end{aligned}$$

where $d = 3$ is the number of spatial dimensions, $\hat{\mathbf{r}}_{ij} = \frac{\mathbf{r}_j - \mathbf{r}_i}{r_{ij}}$, and f_{ij} is the exponent part of the Jastrow factor. The relevant derivatives are for J_1:

$$\begin{aligned}\frac{\partial f_{ij}}{\partial r_{ij}} &= \frac{a}{(1 + r_{ij}\beta)^2} \\ \frac{\partial^2 f_{ij}}{\partial r_{ij}^2} &= -\frac{2a\beta}{(1 + r\beta)^3},\end{aligned}$$

for J_2:

$$\begin{aligned}\frac{\partial f_{ij}}{\partial r_{ij}} &= a\left(\beta_1 + 2r_{ij}\beta_2\right) \\ \frac{\partial^2 f_{ij}}{\partial r_{ij}^2} &= 2a\beta_2,\end{aligned}$$

and finally for J_3:

$$\begin{aligned}\frac{\partial f_{ij}}{\partial r_{ij}} &= \frac{a\left(\beta_1 + 2r_{ij}\beta_2 - \beta_3 + r_{ij}^2\beta_2\beta_3\right)}{(1 + r_{ij}\beta_3)^2} \\ \frac{\partial^2 f_{ij}}{\partial r_{ij}^2} &= \frac{2a\left(\beta_2 + \beta_3\left(-\beta_1 + \beta_3\right)\right)}{(1 + r_{ij}\beta_3)^3}.\end{aligned}$$

6.5.2 Bosons

We again require the first and second derivatives of both the single particle function $G(R)$ and correlation function $F(R)$. We get

$$\begin{aligned}\frac{\nabla_i G}{G} &= \frac{\nabla_i \prod_{j=1}^{N} g(\mathbf{r}_j;\alpha)}{\prod_{j=1}^{N} g(\mathbf{r}_j;\alpha)} \\ &= \frac{\nabla_i g(\mathbf{r}_i;\alpha)}{g(\mathbf{r}_i;\alpha)} \\ &= \frac{\nabla_i \exp\left(-\alpha\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right)}{\exp\left(-\alpha\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right)} \\ &= \frac{\exp\left(-\alpha\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right)}{\exp\left(-\alpha\left(x_i^2 + y_i^2 + \lambda z_i^2\right)\right)}\left(-2\alpha\left(x_i\hat{\mathbf{x}} + y_i\hat{\mathbf{y}} + \lambda z_i\hat{\mathbf{z}}\right)\right) \\ &= -2\alpha\left(x_i\hat{\mathbf{x}} + y_i\hat{\mathbf{y}} + \lambda z_i\hat{\mathbf{z}}\right), \qquad (6.3)\end{aligned}$$

[3]Note that since $r_{ij} = r_{ji}$ and $\hat{\mathbf{r}}_{ij} = -\hat{\mathbf{r}}_{ji}$ we can write the expressions as single sums.

and

$$\begin{aligned}\frac{\nabla_i^2 G}{G} &= \frac{1}{G}\nabla_i\left(G\frac{\nabla_i G}{G}\right)\\ &= \frac{1}{G}\left(G\nabla_i\left(\frac{\nabla_i G}{G}\right)+\left(\frac{\nabla_i G}{G}\right)\nabla_i G\right)\\ &= \nabla_i\left(\frac{\nabla_i G}{G}\right)+\left(\frac{\nabla_i G}{G}\right)^2\\ &= -2\alpha\left(2+\lambda\right)+4\alpha^2\left(x_i^2+y_i^2+\lambda^2 z_i^2\right), \end{aligned} \tag{6.4}$$

moving on to the correlation function we get ([25] page 89)

$$\frac{\nabla_i F}{F} = \sum_{j\neq i}^{N}\frac{a}{r_{ij}\left(r_{ij}-a\right)}\hat{\mathbf{r}}_{ij}, \tag{6.5}$$

and

$$\begin{aligned}\frac{\nabla_i^2 F}{F} &= \sum_{j,k\neq i}^{N}\frac{a}{r_{ij}\left(r_{ij}-a\right)}\frac{a}{r_{ik}\left(r_{ik}-a\right)}\hat{\mathbf{r}}_{ij}\cdot\hat{\mathbf{r}}_{ik}\\ &\quad+\sum_{j\neq i}^{N}\left(\frac{a\left(a-2r_{ij}\right)}{\left(r\left(r-a\right)\right)^2}+\frac{2}{r_{ij}}\frac{a}{r_{ij}\left(r_{ij}-a\right)}\right).\end{aligned} \tag{6.6}$$

Optimizing the Correlation Derivative for the Local Energy

When calculating the local energy the most demanding part is given by the sum

$$\sum_{i=0}^{N}\frac{\nabla_i^2 F}{F}.$$

We see that this is a triple sum over the number of particles, so it has time complexity $\mathrm{O}\left(N^3\right)$. This is the only part of our algorithm with $\mathrm{O}\left(N^3\right)$ time complexity (after the quantum force update optimization discussed in section 6.4.2), which makes it clear that for large systems (i.e. many particles), this sum will dominate the execution time. The way it is given above is the brute force way, however it is possible to make it significantly faster (though ultimately it will still have $\mathrm{O}\left(N^3\right)$ complexity). To condense the notation a bit let us first define

$$f_{ijk} = \frac{a}{r_{ij}\left(r_{ij}-a\right)}\frac{a}{r_{ik}\left(r_{ik}-a\right)}\hat{\mathbf{r}}_{ij}\cdot\hat{\mathbf{r}}_{ik},$$

and note that $f_{ijk} = f_{ikj}$. In this notation the original sum is

$$\sum_{i=0}^{N}\frac{\nabla_i^2 F}{F} = \sum_{i=0}^{N}\left(\sum_{j,k\neq i}^{N} f_{ijk}+\sum_{j\neq i}^{N}\left(\frac{a\left(a-2r_{ij}\right)}{\left(r_{ij}\left(r_{ij}-a\right)\right)^2}+\frac{2}{r_{ij}}\frac{a}{r_{ij}\left(r_{ij}-a\right)}\right)\right).$$

Consider instead the sum

$$\sum_{i=0}^{N} \frac{\nabla_i^2 F}{F} = \sum_{i}^{N} \left[\sum_{j=i+1}^{N} \left[2 \left(\frac{a\left(a - 2r_{ij}\right)}{\left(r\left(r-a\right)\right)^2} + \frac{2}{r_{ij}} \frac{a}{r_{ij}\left(r_{ij}-a\right)} \right) + \sum_{k=j+1}^{N} \left(2f_{ijk} + 2f_{jik} + 2f_{kji}\right) \right] + \sum_{j \neq i}^{N} f_{ijj} \right],$$

which is equivalent. Most notably the sum over k takes care of all of the combinations we would have gotten in the original sum. Multiplying by two takes care of the permutations of the last two indexes since $f_{ijk} = f_{ikj}$. In addition we had to add a sum with the f_{ijj} elements.

While this sum still has $\mathrm{O}\left(N^3\right)$ time complexity it still leads to significantly fewer operations (both memory transfers and calculations). For $N = 100$ this sum is more than three times as fast as the original brute force sum[4]. This specific sum is analyzed further in section 7.5.7, where we count the floating point operations needed more precisely and compare it with our knowledge of the underlying hardware.

6.6 Code Structure

As has been previously mentioned the implementation consists of a C++ host code/program and a set of OpenCL C kernels and functions. The actual Monte Carlo engine is implemented in the kernel code. On the host side almost all OpenCL API calls are handled by the MonteCL class. The full source code of the implementation can be found in ref. [6].

6.6.1 Configuration File

The program is configured through a configuration file, which is interpreted using the program options library of the boost C++ libraries (`http://www.boost.org/`). The configuration file contains information about the physical system (number of particles, wave function, time step and so on), along with details of the simulation (whether to do VMC or DMC, what OpenCL device to use, number of Monte Carlo cycles and so on).

The program options library allows us to read options both from the configuration file and from the command line, meaning that options in the configuration file can be overridden by also specifying them on the command line. As is common practice a help message listing all available options is printed if the program is started with the argument "-h" or "–help", however the only important command is "-c", which takes as an argument the name of the configuration file to be used. The default file is *conf_montecl_default.cfg*.

[4] Specifically on a run with $N = 100$ particles on an AMD Cypress GPU the execution time of the energy kernel went from 2534 ms to 791 ms, with 5120 parallel systems/work-items.

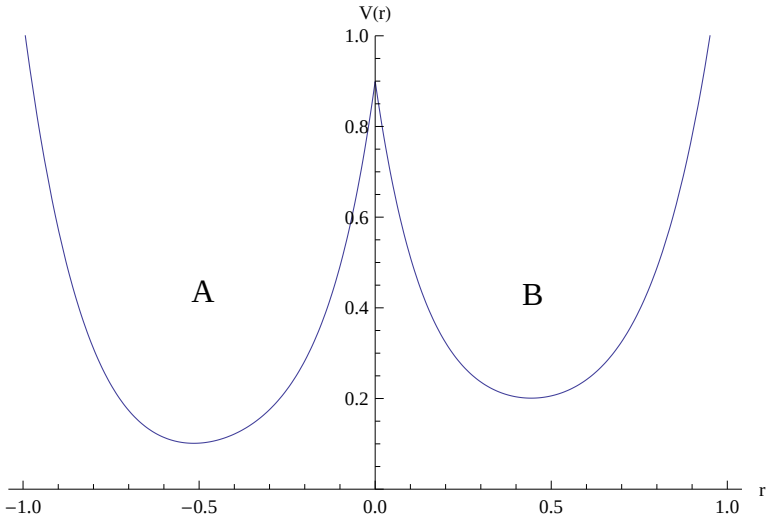


Figure 6.1: Example of a possibly problematic potential for the importance sampled Metropolis Monte Carlo algorithm. A walker could get stuck in the potential wells A or B, leading to poor sampling.

6.6.2 General Description of Implementation

The VMC and DMC algorithms are presented in chapter 3, and specifically algorithms 3.1 and 3.2. The Metropolis Monte Carlo method for VMC is not in itself a good candidate for parallelization, as it is a Markov chain where each step depends on the previous. However there is no reason to perform only a single Monte Carlo simulation.

Indeed it can in certain situations be problematic to restrict the simulation to a single walk. For example consider the hypothetical one-dimensional potential in figure 6.1. Because of the importance sampling that always leads the walker towards the more probable regions (which will usually correspond to minima in the potential), the walker could get "stuck" in either region A or B. Since the Markov chain is ergodic (see section 3.4.2) we know that a walker in region A *can* reach region B, but it could possibly take a long time (i.e. longer than our simulation time). Worse yet, this problem gets amplified when we decrease the time step, which we generally want to do.

Therefore it can be considered advantageous to run several independent simulations (with different initial configurations and random numbers), and then use the results from all of these simulations in our final analysis. This decreases the likelihood of stuck walkers causing problems, and should generally lead to better statistics since we have a large number of (presumably) uncorrelated samples.

The natural way to map this to the OpenCL programming model (chapter 5) is to have each work-item work on an independent system, with its own set

of particles and random number stream. Since each work-item quite naturally winds up performing the same set of calculations on different input data this approach is very SIMD (Single Instruction Multiple Data, see chapter 5) friendly, making it ostensibly suitable for execution on GPUs.

MPI

OpenCL is used to parallelize the computation on a single OpenCL device, however the implementation also makes use of MPI (Message Passing Interface) to further parallelize across several OpenCL devices, either if there are more than one device on a single computer, or to run on a cluster.

For VMC this is very simple since we only have to ensure that the random number streams on the MPI nodes are different. This is simply handled by adding the MPI rank (the unique ID of a MPI process) to the random number seed used both in the host code and in the kernel-side RANLUX generator.

Beyond this there are no complications when doing VMC across several processes. Each process writes its results to file (assuming the *writeenergy* and/or *writepos* options in the configuration file are set) and we simply use all the generated files when calculating the results.

For the DMC case a bit more work is required. After each Monte Carlo cycle the master node (the MPI process with rank 0) gathers all energy values and particle positions from all nodes, and performs the DMC branching algorithm (algorithm 3.2) on the whole set of walkers. Then these walkers are distributed evenly among the processes again and another Monte Carlo cycle is performed.

Check-pointing/Restart

As mentioned the program can write energy values and positions to file for later analysis. It is also the case that the only thing needed to resume a simulation is the energy values and positions (only positions are needed for VMC). Thus it is a simple process to restart from a previously interrupted simulation.

The program writes energy and position files for every Monte Carlo cycle. If the option *vmcresume* or *dmcresume* is enabled in the configuration file the program finds and loads the last files written and uses them to resume the simulation. Since the random number generator state is not stored, it is important to change the seed to the random number generator before restarting. The seed is set by the *prngseed* option in the configuration file.

6.6.3 Data types

When using OpenCL we are transferring variables between two different systems: the host system using C++ and the OpenCL device using OpenCL C. It is important that we have consistent types between the host code and kernel code, and to this end there are special types in the host code that correspond to the types in kernel code. The relevant types are listed in table 6.1. There are also corresponding vector types that are simply defined as type$n$, where n

OpenCL	Host	Description
int	cl_int	Signed 32-bit integer
uint	cl_uint	Unsigned 32-bit integer
long	cl_long	Signed 64-bit integer
ulong	cl_ulong	Unsigned 64-bit integer
float	cl_float	32-bit floating point
double	cl_double	64-bit floating point

Table 6.1: OpenCL data types with their corresponding host code types.

is the number of components in the vector. For example float3 and cl_float3 for a three-component 32-bit floating point vector in kernel and host code, respectively.

The integer types are hard coded, and are usually declared as int or uint. It is also noteworthy that potentially large host-exclusive variables that could overflow a 32-bit integer (like the total number of Monte Carlo cycles) make use of the cl_long and cl_ulong data types, since C++ does not currently have standard portable 64-bit integers[5]. Even though C++ does not have a portable 64-bit integer type, we are guaranteed that any compliant OpenCL implementation will make the cl_long and cl_ulong types available with the appropriate size.

The host code exclusive floating point variables all use the C++ double type (which is 64-bit on most systems). However for the floating point variables that are either exclusive to the OpenCL kernel code, or that are used on both the host and in kernel code (like energies and particle positions) we can choose between the OpenCL types float or double. This is handled through the macro *OCLUSEDOUBLE*, defined in the host code (at the top of the main source file *host_montecl_main.cpp*). If the macro is defined, all floating point variables and calculations are performed with doubles (i.e. 64-bit precision). The only exception is the random number generator which is still generating 32-bit numbers[6].

To facilitate this option we use typedefs, with the host types cl_myfloat_t, cl_myfloat3_t and cl_myfloat4_t for floating point scalar, three-component and four-component vectors respectively, and equivalently myfloat_t, myfloat3_t and myfloat4_t in kernel code. It should be noted that all results presented in chapter 7 are generated using doubles unless otherwise noted.

[5]The next C++ standard (usually referred to as C++0x) will have a 64-bit data type named long long.

[6]We can note that since the integrals we are solving often have hundreds, or even thousands of dimensions, the fact that each dimension (particle coordinate) is influenced by "just" 24 pseudo-random bits is unlikely to give bad results. The set of possible states is still quite astronomical.

on	$r_1^{w_0}$	$r_2^{w_0}$	$\cdots$	$r_N^{w_0}$	$r_1^{w_1}$	$r_2^{w_1}$	$\cdots$	$r_N^{w_1}$	$\cdots$	$r_N^{w_{M-1}}$
off	$r_1^{w_0}$	$r_1^{w_1}$	$\cdots$	$r_1^{w_{M-1}}$	$r_2^{w_0}$	$r_2^{w_1}$	$\cdots$	$r_2^{w_{M-1}}$	$\cdots$	$r_N^{w_{M-1}}$

Figure 6.2: Illustration of memory layout depending on *arraylinear* setting. There are N particles per work-item, and M work-items.

6.6.4 Memory Layout

As mentioned the approach chosen for parallelization is to have different systems simulated by each work-item. This means that if we have for example 5120 work-items and $N = 100$ particles, we need to store $100 \times 5120 = 512000$ vectors. These vectors are stored in a single OpenCL buffer. For this example the total size of this buffer would be a little over 16 MB[7]. We also need a similar buffer for the quantum force, and (for the fermion/atomic case) another set of buffers for the Slater determinants.

As was mentioned in section 5.3, GPUs generally like it when work-items access adjacent memory addresses at the same time. Since it is usually the case that all work-items will be accessing the same particle number in parallel we should organize the memory layout so that particle number i of work-item n is adjacent in memory to particle number i of work-item $n + 1$.

This is handled through the *arraylinear* setting in the configuration file, which is an option to the MonteCL class. If *arraylinear* is enabled then values are stored in the most obvious way, i.e. first all N particles of work-item 0 are stored, then all N particles of work-item 1 and so on. This is not a very good storage pattern for GPUs, as when work-item 0 and 1 each try to access the same particle number the two values are actually separated by $N - 1$ values in memory.

If however the *arraylinear* setting is off then the storage pattern is as described above, so that the two values are adjacent in memory. This is illustrated in figure 6.2. The same approach is applied to the quantum force and Slater determinant buffers.

This switch in storage pattern is handled by small helper functions that provide the actual memory index of a value, given the work-item ID and the index that work-item is trying to access. In the C++ host code it is handled by the following code:

```
size_t ArrIdx(
  size_t index,
  size_t workitem,
  size_t numWorkitems,
  size_t subArraySize,
  bool arrayLinear)
{
  size_t retVal;
```

[7] In OpenCL a three-component vector (for example a double3) is actually stored like a four-component vector (double4).

```
  if(arrayLinear)
    retVal = index + workitem * subArraySize;
  else
    retVal = workitem + index * numWorkitems;

  return retVal;
}
```

And the equivalent code in the OpenCL C source:

```
#ifdef ARRAYLINEAR
#define R_IDX(i) ((i) + get_global_id(0) * NUMPART)
#else
#define R_IDX(i) (get_global_id(0) + (i) * get_global_size(0))
#endif //ARRAYLINEAR
```

Whenever such an array is accessed either in host or kernel code, these helper functions are always used to access the appropriate value.

This approach to accessing memory is actually the only major optimization done for GPGPU, and as we will see it suffices to provide decent performance. The effect of the *arraylinear* setting on performance is studied in section 7.5.4.

6.6.5 The MonteCL Class

The MonteCL class expects to receive an OpenCL context and queue, a couple of other options, along with the name of the configuration file and the argument list. The MonteCL constructor then allocates the needed buffers on the OpenCL device, compiles the OpenCL kernel code and sets everything up so we're ready for simulation.

The main application can then call the InitializeSystem method, generating a random starting state (or alternatively using already defined state, for example previously loaded from a file). The class mostly deals with abstracting the OpenCL operations, for example we have methods for transferring data between host memory and device memory, a method for doing a Monte Carlo cycle and so on. The class doesn't know about VMC or DMC, that is something the main program uses the functionality of the class to implement.

6.6.6 Defining the System

As mentioned in the introduction the implementation is designed to be extendable, although the host-code part is unfortunately not very elegant in this regard at this point. However in kernel code all system-specific code is contained in separate OpenCL C source code files. By systems I mean the two cases of BEC and atoms.

The two relevant files are *kernel_montecl_wavefunction_atomic.cl* and *kernel_montecl_wavefunction_dubois.cl* for the atoms and BEC respectively. Both files contain a standard set of functions that are expected to exist by the kernels

detailed in section 6.6.7. These standard functions should be quite general for other systems suitable for simulation, and so adding a different system would mostly involve coding the corresponding functions into a new source code file.

The functions expected are as follows, with examples presented for the BEC case:

WfRatioSinglePart

Returns the ratio of single particle wave functions with a single particle moved, i.e. $\frac{\psi_S(R')}{\psi_S(R)}$. For example for the BEC case $\frac{G(R')}{G(R)}$, see eq. (6.1). The function takes a parameter i indicating the particle that has been moved. This is implemented as:

```
myfloat_t WfRatioSinglepart(
  global myfloat3_t *r_old_arr,
  myfloat3_t r_new,
  uint i,
  global myfloat_t *SDInvUpOld,
  global myfloat_t *SDInvUpNew,
  global myfloat_t *SDInvDownOld,
  global myfloat_t *SDInvDownNew,
  myfloat4_t varParams)
{
  myfloat_t g;
  myfloat3_t r_old = r_old_arr[R_IDX(i)];

  g = -varParams.x * (r_new.x * r_new.x + r_new.y * r_new.y
    + varParams.y * r_new.z * r_new.z)
    + varParams.x * (r_old.x * r_old.x + r_old.y * r_old.y
    + varParams.y * r_old.z * r_old.z);

  return exp(g);
}
```

WfRatioCorrelated

Returns the ratio of correlation functions $\frac{\psi_C(R')}{\psi_C(R)}$. For example for the BEC case eq. (6.2). The function takes a parameter i indicating the particle that has been moved. This is implemented as:

```
myfloat_t WfRatioCorrelated(
  global myfloat3_t *r_old_arr,
  myfloat3_t r_new,
  uint i,
  myfloat4_t varParams)
{
  myfloat_t f = 1;
```

```
  #ifndef CORRELATION_OFF

  myfloat3_t rj, r_old = r_old_arr[R_IDX(i)];

  for(uint j=0; j<NUMPART; j++){
    if(i != j){
      rj = r_old_arr[R_IDX(j)];
      f *= (1 - varParams.z / distance(rj, r_new))
        / (1 - varParams.z / distance(rj, r_old));
    }
  }

  #endif //CORRELATION_OFF

  return f;
}
```

DelWfSinglepart

Returns $\frac{\nabla_i \psi_S}{\psi_S}$. For the BEC case this is eq. (6.3), implemented as:

```
myfloat3_t DelWfSinglepart(
  myfloat3_t r,
  uint i,
  global myfloat_t *dummy1,
  global myfloat_t *dummy2,
  myfloat4_t varParams)
{
  r.z *= varParams.y;

  return -2 * varParams.x * r;
}
```

Del2WfSinglepart

Returns $\frac{\nabla_i^2 \psi_S}{\psi_S}$. For the BEC case this is eq. (6.4), implemented as:

```
myfloat_t Del2WfSinglepart(
  myfloat3_t r,
  uint i,
  global myfloat_t *dummy1,
  global myfloat_t *dummy2,
  myfloat4_t varParams)
{
  //Del_i^2(G(r_i)) / G
```

```
  return 4 * varParams.x * varParams.x
    * (r.x * r.x + r.y * r.y
    + varParams.y * varParams.y * r.z * r.z)
    - 2 * varParams.x * (2 + varParams.y);
}
```

DelWfCorrelated

Returns $\frac{\nabla_i \psi_C}{\psi_C}$. For the BEC case this is eq. (6.5), implemented as:

```
myfloat3_t DelWfCorrelated(
  global myfloat3_t *r,
  uint i,
  myfloat4_t varParams)
{
  myfloat3_t Result = (myfloat3_t)0.0;
  #ifndef CORRELATION_OFF
  myfloat_t r_ij;
  myfloat3_t ri, rj;
  ri = r[R_IDX(i)];
  for(uint j=0; j<NUMPART; j++){
    if(i != j){
      rj = r[R_IDX(j)];
      r_ij = distance(ri, rj);
      Result += (ri - rj) * (myfloat3_t)(varParams.z
        / (r_ij * r_ij * (r_ij - varParams.z)));
    }
  }
  #else //CORRELATION_OFF
  Result = (myfloat3_t)0.0;
  #endif //CORRELATION_OFF
  return Result;
}
```

Optional/used internally: DelWfCorrelatedSingleInter

Related to DelWfCorrelated above, but returns only the part of the derivative depending on interactions between particles i and j. This is used by the QuantumForceUpdateSingleMove function (described below) for the BEC case where the quantum force optimization presented in section 6.4.2 is used, and is as such only implemented for the BEC case (but something similar could also be interesting for other systems). It is implemented as:

```
myfloat3_t DelWfCorrelatedSingleInter(
  myfloat3_t ri,
  myfloat3_t rj,
  int i,
  int j,
```

```
  myfloat4_t varParams)
{
  myfloat3_t Result = (myfloat3_t)0.0;
  #ifndef CORRELATION_OFF
  myfloat_t r_ij;
  if(i != j){
    r_ij = distance(ri, rj);
    Result += (ri - rj) * (myfloat3_t)(varParams.z
      / (r_ij * r_ij * (r_ij - varParams.z)));
  }
  #else //CORRELATION_OFF
  Result = (myfloat3_t)0.0;
  #endif //CORRELATION_OFF
  return Result;
}
```

Del2WfCorrelatedAllPart

Returns $\sum_{i=1}^{N} \frac{\nabla_i^2 \psi_C}{\psi_C}$. For the BEC case this is the optimized sum presented in section 6.5.2, while for the atomic case it is the equivalent brute force sum. For the BEC case it is implemented as:

```
myfloat_t Del2WfCorrelatedAllPart(
  global myfloat3_t *r,
  myfloat4_t varParams)
{
  myfloat_t del2WfResult = 0;

  #ifndef CORRELATION_OFF

  myfloat_t r_ij, r_ik, r_jk;

  myfloat3_t ri, rj, rk;

  for(uint i=0; i<NUMPART; i++){
    ri = r[R_IDX(i)];

    //The case when j == k
    for(uint jk=0; jk<NUMPART; jk++){
      if(jk != i){
        rj = r[R_IDX(jk)];
        r_ij = distance(ri, rj);
        del2WfResult += varParams.z * varParams.z
          * dot(rj - ri, rj - ri)
          / (r_ij * r_ij * r_ij * r_ij * (r_ij
          - varParams.z) * (r_ij - varParams.z));
      }
    }
```

```
    for(uint j=i+1; j<NUMPART; j++){
      rj = r[R_IDX(j)];
      r_ij = distance(ri, rj);

      //Multiply by two since each iteration also accounts
      //for the case when i and j are switched.
      del2WfResult += 2 * (varParams.z * (varParams.z
        - 2.0 * r_ij) / ( (r_ij * (r_ij - varParams.z))
        * (r_ij * (r_ij - varParams.z)) ) + 2.0
        * varParams.z / (r_ij * r_ij * (r_ij - varParams.z)));

      for(uint k=j+1; k<NUMPART; k++){
        rk = r[R_IDX(k)];
        r_ik = distance(ri, rk);
        r_jk = distance(rj, rk);
        del2WfResult += 2 * (varParams.z * varParams.z
          * dot(ri - rj, ri - rk) / (r_ij * r_ij
          * r_ik * r_ik * (r_ij - varParams.z)
          * (r_ik - varParams.z)));

        //Now switch so that j and i switches places.
        del2WfResult += 2 * (varParams.z * varParams.z
          * dot(rj - ri, rj - rk) / (r_ij * r_ij
          * r_jk * r_jk * (r_ij - varParams.z)
          * (r_jk - varParams.z)));

        //And now k and i switches places instead
        del2WfResult += 2 * (varParams.z * varParams.z
          * dot(rk - rj, rk - ri) / (r_jk * r_jk
          * r_ik * r_ik * (r_jk - varParams.z)
          * (r_ik - varParams.z)));
      }
    }
  }

  #else //CORRELATION_OFF

  del2WfResult = 0;

  #endif //CORRELATION_OFF

  return del2WfResult;
}
```

PotentialEnergy

Returns the total potential energy (summed for all particles). For the BEC case it is eq. (4.1), implemented as:

```
myfloat_t PotentialEnergy(
  global myfloat3_t *r,
  myfloat4_t varParams)
{
  //Potential energy (harm. osc. trap)
  myfloat_t pot = 0;
  myfloat3_t ri;
  for(uint i=0; i<NUMPART; i++){
    ri = r[R_IDX(i)];
    ri = ri * ri;
    pot += ri.x + ri.y + varParams.y * varParams.y * ri.z;
  }

  return pot * 0.5;
}
```

QuantumForceUpdateSingleMove

Updates the quantum force when only a single particle has been moved. For the BEC case this uses the optimization discussed in section 6.4.2, while for the atoms it is done in a brute force manner (recalculating the quantum force completely). For the BEC case it is implemented as:

```
void QuantumForceUpdateSingleMove(
  global myfloat3_t *r,
  myfloat3_t rNew,
  uint particleNr,
  global myfloat3_t *QfOld,
  global myfloat3_t *QfNew,
  global myfloat_t *dummy1,
  global myfloat_t *dummy2,
  myfloat4_t varParams)
{
  //Updates the quantum force, given that only
  //particle <particleNr> has been moved, and
  //rNew is the new position of that particle.

  myfloat3_t result;

  for(uint i=0; i<NUMPART; i++){
    result = 0.5 * QfOld[QF_IDX(i)];

    if(i == particleNr){
      for(uint j=0; j<NUMPART; j++){
        result -= DelWfCorrelatedSingleInter
          (r[R_IDX(i)], r[R_IDX(j)], i, j, varParams);
        result += DelWfCorrelatedSingleInter
          (rNew, r[R_IDX(j)], i, j, varParams);
      }
```

```
    }

    result -= DelWfCorrelatedSingleInter
      (r[R_IDX(i)], r[R_IDX(particleNr)],
       i, particleNr, varParams);
    result += DelWfCorrelatedSingleInter
      (r[R_IDX(i)], rNew, i, particleNr, varParams);

    if(i == particleNr){
      result -= DelWfSinglepart
        (r[R_IDX(i)], i, dummy1, dummy2, varParams);
      result += DelWfSinglepart
        (rNew, i, dummy1, dummy2, varParams);
    }

    QfNew[QF_IDX(i)] = 2.0 * result;
  }
}
```

NewPositionAllowed

Checks whether a new particle position is admissible or not. For the BEC case it checks that the new position isn't within the scattering length a of another particle, effectively implementing eq. (4.2). For the atomic case it just checks that an electron does not stray extremely close to either the nucleus or other atoms (to avoid precision problems). For the BEC case it is implemented as:

```
bool NewPositionAllowed(
  global myfloat3_t *r,
  myfloat3_t r_new,
  uint particleNr,
  myfloat4_t varParams)
{
  //Make sure particle is not within the hard core radius
  //(scattering length) of any other particle.
  bool tooClose = 0;

  #ifndef CORRELATION_OFF

  myfloat3_t distSqr;

  for(uint i=0; i<NUMPART; i++){
    if(i != particleNr){
      distSqr = r_new - r[R_IDX(i)];
      distSqr = distSqr * distSqr;
      if(distSqr.x + distSqr.y + distSqr.z
        <= varParams.z * varParams.z){
        tooClose = 1;
        break;
```

```
      }
    }
  }

  #endif //CORRELATION_OFF

  return !tooClose;
}
```

This concludes the list of the kernel functions necessary to define a system.

6.6.7 The General Kernels and Functions

As mentioned previously, the actual Monte Carlo machinery is implemented in OpenCL C kernels, and this machinery is administered by the host code. The kernels and functions presented here are the same for both the atoms and BEC and are found in the kernel_montecl.cl source code file. The kernels are (generally in the order they are launched):

RanluxclWarmup

Implements the RANLUX warmup function mentioned in section 6.1.1. It is only launched once by the MonteCL class constructor to ensure decorrelated numbers between the work-items. It is implemented as:

```
__kernel void RanluxclWarmup(global float4* RANLUXCLTab)
{
  ranluxcl_state_t ranluxclstate;
  ranluxcl_download_seed(&ranluxclstate, RANLUXCLTab);
  ranluxcl_warmup(&ranluxclstate);
  ranluxcl_upload_seed(&ranluxclstate, RANLUXCLTab);
}
```

QfInitialization

Performs a full calculation of the quantum force (i.e. not assuming that the quantum force has been calculated previously, so it does not use the optimization mentioned in section 6.4.2), and is called every time we make a call to the DoMCCycles method of the MonteCL class. It is implemented as:

```
__kernel void QfInitialization(
  private myfloat4_t varParams,
  global myfloat3_t *r_old,
  global myfloat3_t *QF_old,
  global myfloat_t *SDInvUpOld,
  global myfloat_t *SDInvDownOld)
{
  //Calculate initial quantum force
  QuantumForceUpdateFull
```

```
    (r_old, QF_old, SDInvUpOld, SDInvDownOld, varParams);
}
```

NewPosition

Generates a new trial position for a single particle using eq. (3.3), implemented as:

```
__kernel void NewPosition(
  private uint particleNr,
  private myfloat4_t varParams,
  global float4 *RANLUXCLTab,
  global myfloat3_t *r_old,
  global myfloat3_t *r_new,
  global myfloat3_t *QF_old,
  global float *RandomNumber)
{
  uint gid = get_global_id(0);

  myfloat3_t r_new_temp;
  float4 randomvecUniform;
  myfloat3_t randomvecNormal;

  ranluxcl_state_t ranluxclstate;
  ranluxcl_download_seed(&ranluxclstate, RANLUXCLTab);

  do{
    randomvecUniform = ranluxcl(&ranluxclstate);
    randomvecNormal = (NormalDist(randomvecUniform)).xyz;
    r_new_temp = r_old[R_IDX(particleNr)] + randomvecNormal
      * (myfloat3_t)(sqrt(TIMESTEP))
      + QF_old[QF_IDX(particleNr)]
      * (myfloat3_t)(TIMESTEP * D);
  }while(!NewPositionAllowed
              (r_old, r_new_temp, particleNr, varParams));

  r_new[gid] = r_new_temp;
  RandomNumber[gid] = randomvecUniform.w;

  ranluxcl_upload_seed(&ranluxclstate, RANLUXCLTab);
}
```

RatiosAndGf

Computes the wave function ratio and Green's function ratio (see section 6.2.1) needed in the metropolis algorithm. It is implemented as:

```
__kernel void RatiosAndGf(
  private uint particleNr,
```

```
  private myfloat4_t varParams,
  global myfloat3_t *r_old,
  global myfloat3_t *r_new,
  global myfloat3_t *QF_old,
  global myfloat3_t *QF_new,
  global myfloat_t *SDInvUpOld,
  global myfloat_t *SDInvUpNew,
  global myfloat_t *SDInvDownOld,
  global myfloat_t *SDInvDownNew,
  global myfloat_t *MetropolisWeight)
{
  uint gid = get_global_id(0);
  myfloat_t WFRatio;

  //Wave function ratios. WfRatioSinglepart also updates
  //the SD matrixes for the atomic case.
  WFRatio = WfRatioSinglepart
    (r_old, r_new[gid], particleNr, SDInvUpOld,
    SDInvUpNew, SDInvDownOld, SDInvDownNew, varParams);
  WFRatio *= WfRatioCorrelated
    (r_old, r_new[gid], particleNr, varParams);

  QuantumForceUpdateSingleMove
    (r_old, r_new[gid], particleNr, QF_old, QF_new,
    SDInvUpNew, SDInvDownNew, varParams);

  myfloat_t Greensfunction = 0;

  for(uint ii=0; ii<NUMPART; ii++){
    if(ii == particleNr)
      Greensfunction += 0.5 * dot(QF_old[QF_IDX(ii)]
        + QF_new[QF_IDX(ii)], (QF_old[QF_IDX(ii)]
        - QF_new[QF_IDX(ii)])
        * (myfloat3_t)(D * TIMESTEP * 0.5)
        - r_new[gid] + r_old[R_IDX(ii)]);
    else
      Greensfunction += 0.5 * dot(QF_old[QF_IDX(ii)]
        + QF_new[QF_IDX(ii)], (QF_old[QF_IDX(ii)]
        - QF_new[QF_IDX(ii)])
        * (myfloat3_t)(D * TIMESTEP * 0.5));
  }

  MetropolisWeight[gid]
    = exp(Greensfunction) * WFRatio * WFRatio;
}
```

MetropolisTest

Performs the Metropolis test, and updates/reverts values based on whether the move proposed by the NewPosition kernel was allowed or not. The Metropolis test is given by eq. (3.5), and is implemented as:

```
__kernel void MetropolisTest(
  private uint particleNr,
  global myfloat3_t *r_old,
  global myfloat3_t *r_new,
  global myfloat3_t *QF_old,
  global myfloat3_t *QF_new,
  global myfloat_t *SDInvUpOld,
  global myfloat_t *SDInvUpNew,
  global myfloat_t *SDInvDownOld,
  global myfloat_t *SDInvDownNew,
  global myfloat_t *MetropolisWeight,
  global float *RandomNumber)
{
  uint gid = get_global_id(0);

  //The Metropolis test is performed by moving one particle
  //at a time
  if(RandomNumber[gid] < MetropolisWeight[gid]){
    //Accepting move

    //Updating coordinates and Quantum Force
    r_old[R_IDX(particleNr)] = r_new[gid];

    for(uint j=0; j<NUMPART; j++)
      QF_old[QF_IDX(j)] = QF_new[QF_IDX(j)];

    #ifdef USINGSDINV
    //Updating SD matrixes
    if(particleNr < NUMPARTUP)
      for(uint j=0; j<NUMPARTUP; j++)
        for(uint k=0; k<NUMPARTUP; k++)
          SDInvUpOld[SDINVUPIDX(j,k)]
            = SDInvUpNew[SDINVUPIDX(j,k)];
    else
      for(uint j=0; j<NUMPARTDOWN; j++)
        for(uint k=0; k<NUMPARTDOWN; k++)
          SDInvDownOld[SDINVDOWNIDX(j,k)]
            = SDInvDownNew[SDINVDOWNIDX(j,k)];
    #endif //USINGSDINV
  }

  else{
    //Rejecting move
    #ifdef USINGSDINV
```

```
    //revert SD matrixes
    if(particleNr < NUMPARTUP)
      for(uint j=0; j<NUMPARTUP; j++)
        for(uint k=0; k<NUMPARTUP; k++)
          SDInvUpNew[SDINVUPIDX(j,k)]
            = SDInvUpOld[SDINVUPIDX(j,k)];
    else
      for(uint j=0; j<NUMPARTDOWN; j++)
        for(uint k=0; k<NUMPARTDOWN; k++)
          SDInvDownNew[SDINVUPIDX(j,k)]
            = SDInvDownOld[SDINVUPIDX(j,k)];
    #endif //USINGSDINV
  }
}
```

Energy

Calculates the local energy. It is implemented as:

```
__kernel void Energy(
  private myfloat4_t varParams,
  private uint Offset,
  global myfloat3_t *r_old,
  global myfloat_t *SDInvUpOld,
  global myfloat_t *SDInvDownOld,
  global myfloat_t *EnergyAllValues)
{
  uint gid = get_global_id(0);

  myfloat_t e = LocalEnergy
    (r_old, SDInvUpOld, SDInvDownOld, varParams);

  EnergyAllValues[gid + Offset * get_global_size(0)] = e;
}
```

When we have N particles, the three kernels NewPosition, RatiosAndGf and MetropolisTest are called N times per Monte Carlo cycle, and the Energy kernel is only launched once at the end. Alternatively, for the VMC algorithm we can elect not to launch the Energy kernel every time, as for the BEC case it is the most computationally intensive. It can therefore pay to do several Monte Carlo cycles between each call to the Energy kernel, as this means the energy values will not be as strongly correlated.

We additionally have the following helper functions that are general and are used by the above kernels. They are:

NormalDist

Implements the Box-Muller transform described in section 6.1.2. It is implemented as:

```
myfloat4_t NormalDist(float4 U)
{
  //Returns a vector where each component is a normally
  //distributed PRN centered on 0, with standard deviation
  //1. Note: M_PI_F is an OpenCL macro for the value of pi.
  //M_PI would be the 64-bit double version.
  myfloat4_t Z;
  float R, phi;

  R = sqrt(-2 * log(U.x));
  phi = 2 * M_PI_F * U.y;
  Z.x = R * cos(phi);
  Z.y = R * sin(phi);

  R = sqrt(-2 * log(U.z));
  phi = 2 * M_PI_F * U.w;
  Z.z = R * cos(phi);
  Z.w = R * sin(phi);

  return Z;
}
```

LocalEnergy

Computes the total local energy of the system. It is implemented as:

```
myfloat_t LocalEnergy(
  global myfloat3_t *r,
  global myfloat_t *SDInvUp,
  global myfloat_t *SDInvDown,
  myfloat4_t varParams)
{
  //Local energy for both fermion and boson systems.

  myfloat_t kinetic = 0;

  kinetic -= Del2WfCorrelatedAllPart(r, varParams);
  for (int i = 0; i < NUMPART; i++){
    kinetic -= Del2WfSinglepart
      (r[R_IDX(i)], i, SDInvUp, SDInvDown, varParams);

    //Remaining kinetic part:
    //2 * del(psi_c) * del(psi_d) / (psi_c * psi_d)
    kinetic -= 2*dot(
      DelWfSinglepart(r[R_IDX(i)], i , SDInvUp,
        SDInvDown, varParams),
      DelWfCorrelated(r, i, varParams));
  }
```

```
  return PotentialEnergy(r, varParams) + kinetic * 0.5;
}
```

QuantumForceUpdateFull

Uses the system-specific derivative functions to do a complete calculation of the quantum force of each particle. The quantum force of a particle is given by eq. (3.4). It is implemented as:

```
void QuantumForceUpdateFull(
  global myfloat3_t *r,
  global myfloat3_t *QF,
  global myfloat_t *SDInvUp,
  global myfloat_t *SDInvDown,
  myfloat4_t varParams)
{
  //Updates the QF array by completely recalculating the
  //quantum force.
  for(uint i=0; i<NUMPART; i++){
    QF[QF_IDX(i)] = 2 * (
      DelWfSinglepart
        (r[R_IDX(i)], i, SDInvUp, SDInvDown, varParams)
      + DelWfCorrelated(r, i, varParams));
  }
}
```

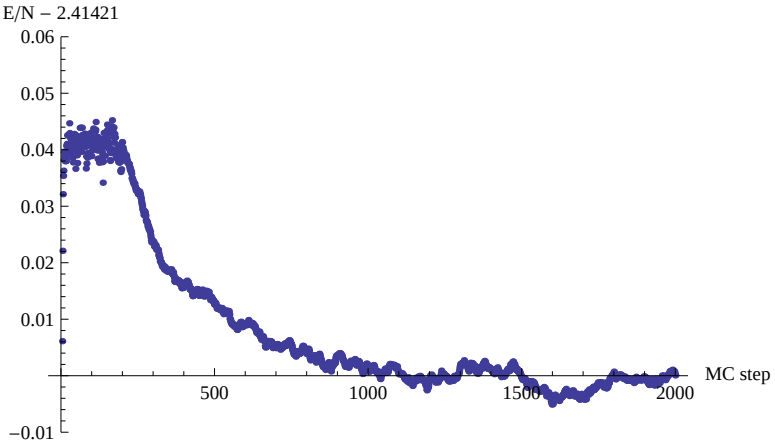
6.7 Testing the Implementation

A simple way to check the correctness of the implementation is to turn off the correlation parts of the computation, so that in the atomic case we are studying non-interacting electrons and in the BEC case non-interacting bosons. In these cases we have the exact wave function and expect to find the correct energy with zero variance in the result (obviously this check does not verify the correlation parts of the algorithm since those parts are disabled). The correlation effects can be turned off for both the atomic and BEC cases by enabling the *correlationoff* option in the configuration file of the program.

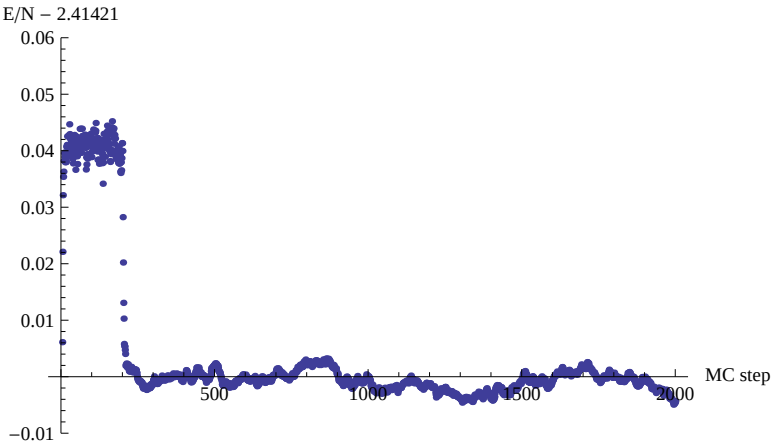
For the BEC system, when correlation effects are turned off and the harmonic oscillator trap is deformed with $\lambda = \sqrt{8}$, the exact wave function is obtained when the variational parameter is $\alpha = 1/2$, and the energy should then be $E/N = 2.41421\,\hbar\omega_\perp$ ([25] section 5.7.2). If the simulation is run with these parameters, the exact energy will be produced with essentially zero variance.

Knowing the exact result also allows us to test the DMC implementation. Even if we start the simulation with a slightly wrong trial wave function, the DMC method should lead us to the exact energy.

Figure 6.3a shows how the DMC algorithm is able to find the correct ground state energy. The plot is with a variational parameter $\alpha = 0.6$ (whereas the



(a) First 200 cycles are VMC, the rest are DMC. Run with $\Delta t = 0.001$ for both VMC and DMC.



(b) Similar to figure 6.3a, however here the first 10 DMC cycles use a long time step of $\Delta t = 0.1$ to speed up thermalisation.

Figure 6.3: Shows how the DMC algorithm finds the correct ground state energy even when the trial wave function is not correct. Run with $\alpha = 0.6$. Energies are in units of $\hbar\omega_\perp$.

exact wave function is obtained with $\alpha = 0.5$). The first 200 cycles are VMC, which gives us an energy that is about 1.5 % above the correct value. The DMC algorithm however finds the correct energy. The simulation was run with a time step $\Delta t = 0.001$, and so we see that DMC used about 1000 cycles (meaning one time unit since $1000 \times 0.001 = 1$) to thermalize.

For larger systems this rather long thermalisation time is undesirable. It would seem like a good idea then to use a larger time step for the thermalisation. Figure 6.3b shows this. It is the same run as figure 6.3a, except that the first 10 DMC cycles are done with a time step $\Delta t = 0.1$. We see that this ostensibly allows us to start sampling the ground state much quicker, saving precious simulation time. This trick will be very helpful for large systems.

Chapter 7

Results

In this chapter the implementation presented in chapter 6 is used to generate results, checking them against values from other work. We also look at the performance, comparing it to other implementations in C++ using the same basic algorithms as used in this thesis.

7.1 Time Step Extrapolation

The algorithm we are using has a time step Δt, and as is often the case there is a time step error, so that in principle we would like to have $\Delta t = 0$. However as we reduce Δt the particles are moved more slowly, and thus we require more Monte Carlo cycles to get a good representative sample. Instead of spending huge amounts of computational time on very small time steps, we instead find results for a few different values for Δt and extrapolate to $\Delta t = 0$.

The actual extrapolation is done by a Fortran program based on the method described in chapter 15.4 of ref. [31]. It performs a linear model fit which extrapolates both the energy and error (based on the errors of the given data points) to $\Delta t = 0$.

When gathering data for different time steps I have preferred to gather data over the same amount of time units, not the same number of Monte Carlo cycles. For example for $\Delta t = 0.001$ I would use ten times as many Monte Carlo cycles as for $\Delta t = 0.01$. The exception to this is for the BEC system with $N = 500$ particles, where time constraints led me to use fewer samples. However in the VMC case I have many independent systems (I've used 5120 systems), so we have at least 5120 uncorrelated energy measurements anyway.

We observe that thermalization usually takes around one time unit (both for VMC from an initial random state to equilibrium and for the transition from VMC to DMC as we saw in figure 6.3a), and thus we can guess that samples separated by more than one time unit will be reasonably decorrelated. Also, looking at figure 6.3a and 6.3b we see that the DMC sample has something akin to a low-frequency variation that has a wave-length of about one time

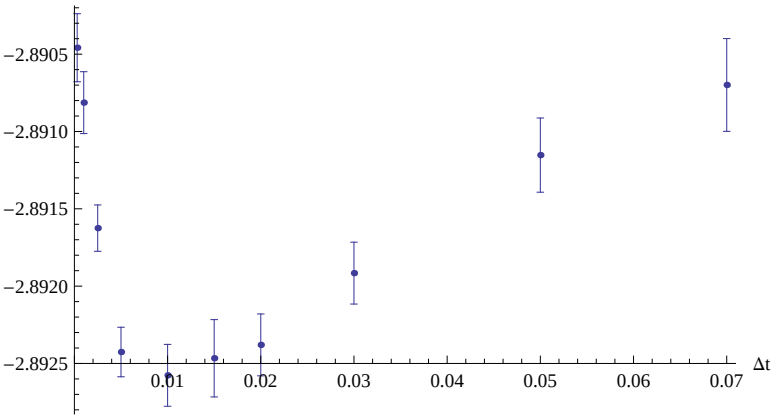


Figure 7.1: Time step extrapolation data for helium. Energy is in units of $E_h = 27.2$ eV, unit of time is arbitrary.

unit. Using just one time unit or less when gathering DMC data we may be unlucky and sample just a peak, giving us a biased measurement which would not be reflected in the error estimate. Therefore when gathering DMC data I always use at least a few time units, preferably even more, though how many are practical of course depends on the system size.

Figures 7.1, 7.2 and 7.3 show the time step data for helium, beryllium and neon respectively. We see that the time step dependence of the energy has a bend at around $\Delta t = 0.01$ for all three atoms (and neon has another bend at even shorter time steps). I do not know why this is, nor have I found similar results in other sources, but as we will see the energies seem to be correct.

In figures 7.4, 7.5 and 7.6 the extrapolation data for the BEC case with $N = 20$ is given. The interesting point here is that the dependence seems to be approximately linear (especially figure 7.5 has many points confirming this for DMC), and so we can hope that the same is the case for $N = 500$ particles, where the large system size means we will not have as many points, nor as short time steps.

7.2 Finding Optimal Variational Parameters

The simplest way to find the optimum values for the variational parameters is to simply compute the energy for several different values, then for example make a surface plot (in the case of two parameters) of the energy and find the minimum this way. The implementation presented here can generate a list of variational parameters and the associated energies by setting the *varparamxstart*, *varparamxend* and *varparamxsteps* options in the configuration file, however we will mostly be relying on results from other sources, and compare our findings with them.

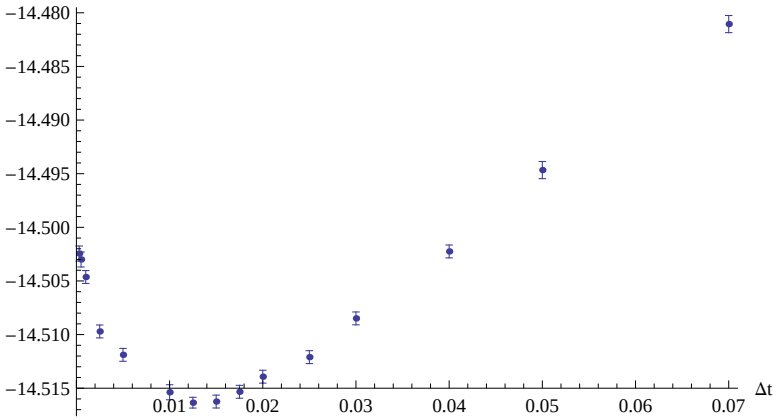


Figure 7.2: Time step extrapolation data for beryllium. Same units as in figure 7.1.

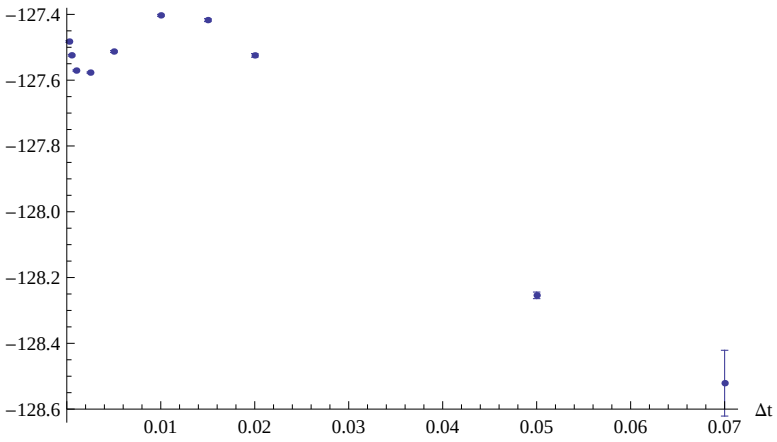


Figure 7.3: Time step extrapolation data for neon. Same units as in figure 7.1.

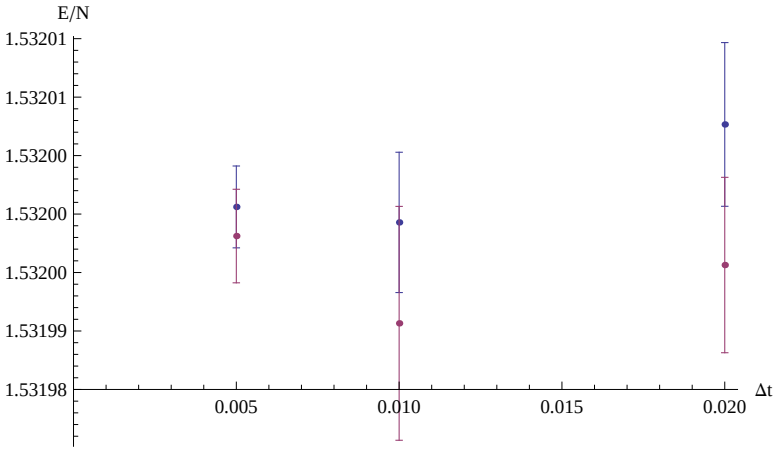


Figure 7.4: Time step extrapolation data for BEC with $N = 20$ particles, scattering length $a = 0.00433a_\perp$ (i.e. $a = a_{\mathrm{Rb}}$) and $\lambda = 1$. Top (blue) is VMC, bottom is DMC. The system is highly dilute and we see that there is virtually no dependence on time step length. Extrapolated energies are given in table 7.2. Energy is in units of $\hbar\omega_\perp$, unit of time is arbitrary.

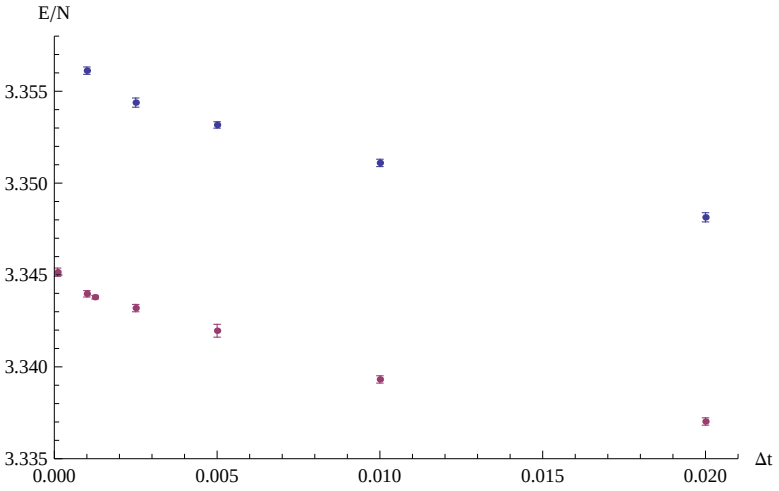


Figure 7.5: Time step extrapolation data for BEC with $N = 20$ particles, scattering length $a = 0.433a_\perp$ (i.e. $a = 100a_{\mathrm{Rb}}$) and $\lambda = 1$. Top (blue) is VMC, bottom is DMC. The system is somewhat dense and we see a larger correction by DMC than we did in figure 7.4. Extrapolated energies are given in table 7.2. Same units as in figure 7.4.

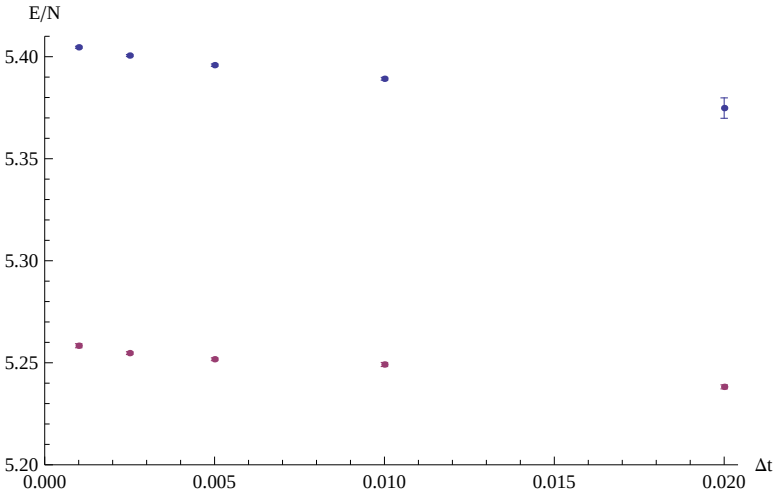


Figure 7.6: Time step extrapolation data for BEC with $N = 20$ particles, scattering length $a = 0.433a_\perp$ (i.e. $a = 100a_{\mathrm{Rb}}$) and $\lambda = \sqrt{8}$. Top (blue) is VMC, bottom is DMC. Because of the deformation the trap is somewhat smaller than that of figure 7.5 (where $\lambda = 1$). However the correction by DMC still seems much more significant. Same units as in figure 7.4.

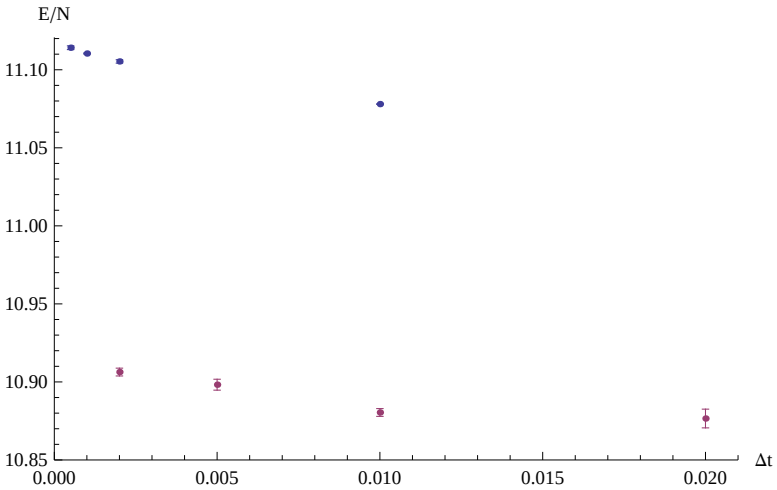


Figure 7.7: Time step extrapolation data for BEC with $N = 500$ particles, scattering length $a = 0.15155a_\perp$ (i.e. $a = 35a_{\mathrm{Rb}}$) and $\lambda = \sqrt{8}$. Top (blue) is VMC, bottom is DMC. The system is quite dense and there is a large correction by DMC. Energies are given in table 7.2. Same units as in figure 7.4.

System	α	β	E_{VMC}	$E_{\mathrm{VMC}}^{\mathrm{Literature}}$
Helium	1.839	0.348	-2.8903(2)	-2.89040(3.9)
Beryllium	3.925	0.109	-14.5015(9)	-14.50220(1.8)
Neon	9.546	0.177	-127.4598(30)	-127.284(2.5)

Table 7.1: Ground state energy results for atomic systems with hydrogenic basis functions and Jastrow 1. VMC reference is ref. [29]. Energy is in units of E_h.

7.3 Atomic Systems

Here the results for the atomic systems are presented. We look at helium, beryllium and neon.

7.3.1 How Data is Gathered

In the interest of reproducibility it is important to present how data for the different systems are gathered.

In all cases I have used 5120 work-items (i.e. 5120 independent systems). For all atomic systems I have used 10 time units for thermalization and 100 time units for data gathering for each work-item for each time step value. For example with $\Delta t = 0.01$ this means 10000 Monte Carlo cycles for thermalization and 100000 cycles for data gathering. Since there are 5120 systems simulated in parallel this gives a total of 512 million data cycles (i.e. energy samples) for this specific time step.

The final energies given in table 7.1 are found by using the three shortest time steps in the time step extrapolation plots (figures 7.1, 7.2 and 7.3), and extrapolating to $\Delta t = 0$.

7.3.2 Energies

Energy results for the systems are presented in table 7.1. The optimal variational parameters for the hydrogen basis with Jastrow 1 (introduced in section 4.1) are taken from table 8.5 in ref. [29]. We see that for helium and beryllium we have very good agreements, while for neon there is some discrepancy.

It should be noted that Sandsdalen (ref. [29]) did not present as many (nor as short) time step lengths as I have in figures 7.1, 7.2 and 7.3, however judging by the values that were presented he did not experience the same bends in time step dependence as I did. Whether this is indicative of a bug in my code or not is uncertain. I am clearly finding reasonable values, and as we will see the code is also generating seemingly good values for the BEC case.

7.4 Bose-Einstein Condensate

For the BEC (Bose-Einstein Condensate) we can compare the implementation with values from refs. [13, 24, 25].

7.4.1 How Data is Gathered

Just as I did for the atomic case I have used 5120 work-items. For DMC the number of walkers (i.e. "active" work-items) was kept within 95 % of the total number of work-items (by setting the *minworkitemsfract* setting to 0.95 in the configuration file).

For the small system with $N = 20$ particles I have used two initial time units for thermalization, and always used four time units for data. When switching time step lengths I have allowed one time unit for thermalization. This was done for both VMC and DMC.

For the larger system with $N = 500$ particles the same amount of thermalization was used, however for the VMC case some of the lower time step samples use fewer data cycles (i.e. less than four time units) to save some execution time. This should be unproblematic since the 5120 work-items give us uncorrelated samples anyway. In the DMC case the branching algorithm introduces correlations, which makes it much more important to let the simulation run for a few time units to ensure we have a proper sample. Therefore the DMC results are gathered using four time units even for the short time steps.

The final energies are based on the three shortest time steps in the time step extrapolation plots (figures 7.4, 7.5, 7.6 and 7.7), extrapolated to $\Delta t = 0$.

7.4.2 Energies

The obtained energies along with reference values from different sources are listed in table 7.2. Energies are given per particle, and we recall that for the non-interacting case the energy when $\lambda = 1$ (spherical trap) is $E/N = 1.5\,\hbar\omega_\perp$ and for $\lambda = \sqrt{8}$ (disk-shaped trap) it is $E/N = 2.41421\,\hbar\omega_\perp$. The less dilute/more dense the system the higher we can expect the energy to be above the non-interacting values.

Results are presented for both the VMC and DMC methods, and we should keep in mind that the DMC results are in principle exact, and can therefore be considered a reference for the VMC results. It the VMC result agrees well with the DMC result we can take it as in indication that our trial wave function models the true ground state wave function quite well.

For few particles and normal scattering length ($N = 20$ and $a = a_{\mathrm{Rb}} = 0.00433\,a_\perp$) we see no difference between the VMC and DMC methods. This is not surprising since this system is so dilute that the interactions only have a minor effect, meaning we are very close to having the exact wave function. When the scattering length is increased by a factor of 100 to $a = 0.433\,a_\perp$, DMC yields a slightly lower energy, but the VMC approach is still very good. We can note that in both cases our DMC results equal (within statistical errors) those of Blume in ref. [13].

In addition the dense case with $a = 100\,a_{\mathrm{Rb}}$ is also studied with $\lambda = \sqrt{8}$, and we see that there is a much larger correction by DMC in this case. For $\lambda = 1$ the correction is just 0.3 %, while for $\lambda = \sqrt{8}$ it is almost 3 %. Though the system is denser (since $\lambda > 1$ compacts the potential in the z-direction), the

N	λ	a ($a_\perp$)	α	E_{VMC}/N	$E_{\mathrm{VMC}}^{\mathrm{reference}}/N$
20	1	0.00433	0.495	1.531998(5)	
20	1	0.433	0.46	3.35664(23)	
20	$\sqrt{8}$	0.433	0.45	5.40649(59)	
500	$\sqrt{8}$	0.15155	0.7687	11.116(1)	11.12109(14)

(a) VMC results.

N	λ	a ($a_\perp$)	E_{DMC}/N	$E_{\mathrm{DMC}}^{\mathrm{reference}}/N$
20	1	0.00433	1.531999(6)	1.53195(5)
20	1	0.433	3.34524(24)	3.345(5)
20	$\sqrt{8}$	0.433	5.259(1)	
500	$\sqrt{8}$	0.15155	10.9132(30)	

(b) DMC results.

Table 7.2: Ground state energy results for BEC systems. VMC reference is ref. [24] and DMC reference is ref. [13]. Energies are in units of $\hbar\omega_\perp$.

larger correction may also be an indication that the VMC approach (or more precisely our trial wave function) does not handle deformed traps quite as well as it does spherical traps.

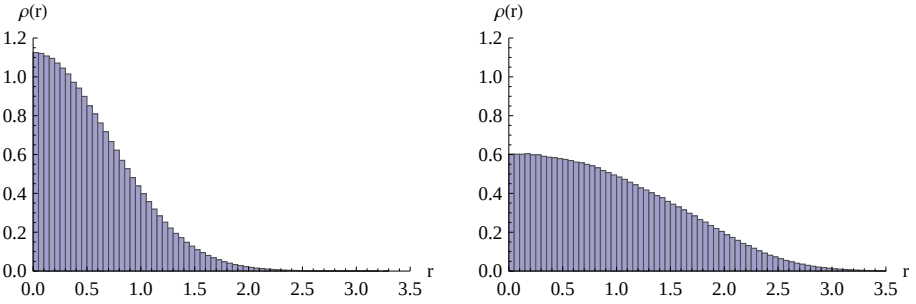
The truly time-consuming run is the one for $N = 500$ particles with deformed trap ($\lambda = \sqrt{8}$) and scattering length $a = 35\, a_{\mathrm{Rb}} = 0.15155\, a_\perp$. The VMC results do not completely line up with those in ref. [24], however they do line up reasonably well with an earlier result by the same author in ref. [25] where he found $E_{\mathrm{VMC}}/N = 11.1169(24)\,\hbar\omega_\perp$. Since there is no information about the time steps used in ref. [24] it is difficult to try to reproduce the result.

For the DMC analysis of the 500-particle system I have found no good source to compare with. There is a DMC run for the system presented in ref. [25], however the author himself notes that there was not enough time to do a proper simulation[1]. The energy found there was $E_{\mathrm{DMC}}/N = 11.0961(69)\,\hbar\omega_\perp$, which is much too high an energy according to my results. Since that result is highly suspect (as the author himself notes) I have not included it in table 7.2.

7.4.3 Particle Distribution

In table 7.2 we saw that the energy of the systems increases well above the non-interacting ground-state values (i.e. that of a plain harmonic oscillator) when the scattering length is increased. Recall that the interaction for the BEC, given by eq. (4.2), is a hard core potential. When the scattering length a (the hard core diameter of a particle) is increased, there simply isn't room for all of the particles when they all try to occupy the ground state at the same time. The

[1] Indeed the DMC run of ref. [25] was based on only a few thousand samples of the local energy, while the result presented here is based on tens of millions of samples.



(a) $a = 0\,a_{\mathrm{Rb}}$, i.e. plain non-interacting harmonic oscillator.

(b) $a = 100\,a_{\mathrm{Rb}}$

Figure 7.8: Histograms showing the particle density distribution for a BEC with $N = 20$ and $\lambda = 1$. The length unit is $a_\perp$.

natural conclusion then is that the particles must move further away from the center of the potential.

Figure 7.8 shows the particle distributions for the BEC with $N = 20$ particles and spherically symmetric trap (meaning $\lambda = 1$). The density histograms are simply histograms of the x-coordinate of the particles (which is sufficient in the spherically symmetric case). The histograms are then normalized as probability distributions (i.e. so that they sum up to 1). We see that the intuitive explanation is correct, the particle distribution is pushed out in space when the system gets denser.

7.5 Performance

Here the performance of the OpenCL implementation will be analyzed, comparing it to other similar known implementations. Such comparisons are not easy to do conclusively, as there is always the question of how well optimized the implementations are. The fact of the matter is that I am no expert in software optimization, meaning that both the reference implementations and my OpenCL implementation could quite likely be further optimized.

The approach taken here is simply to compare to other "similar" implementations, in the sense that the C/C++ implementations I compare with could be developed with a similar skill-set as my own. I cannot claim that the findings presented here are definitive, however the algorithms used in the chosen references are very similar to those I have used. A notable exception is that the reference CPU implementations tend to keep some tables of values to avoid unnecessary re-computations, however that should generally work in their favor in terms of performance.

The reference implementations are run single-threaded (i.e. on a single CPU core), while the OpenCL implementation is run on all six cores of the CPU and

all compute units of the GPU. If anything this should be an advantage to the reference implementations, since they will have all shared resources on the CPU allocated to them.

7.5.1 System Configuration

Unless otherwise noted all code is run on a Windows 7 computer with an AMD Phenom II 1090T 3.2 GHz six-core CPU, and an AMD/ATI Radeon HD 5870 graphics card (codenamed Cypress) with Catalyst 11.5 drivers. Where applicable the code is compiled using Visual Studio 2010 with release settings, but it has also been verified that performance is comparable when compiled in Linux using GCC with the -O3 optimization option. The OpenCL implementation used is the AMD APP SDK version 2.4 [3].

7.5.2 Understanding the Reported Values

When listing the performance figures I use Monte Carlo cycles per second (MC/s). When discussing performance I always equate an energy sample with a Monte Carlo cycle unless otherwise noted. Recall that for the VMC approach we do not actually have to take energy samples for each cycle.

In the case of OpenCL with for example 5120 work-items, when all work-items have each performed a Monte Carlo cycle in parallel that counts as 5120 cycles, since that's how many energy samples were generated. For the single threaded reference implementations only a single walker would be running on a single CPU core, and so it would have to perform 5120 cycles serially to do the same amount of work.

The OpenCL implementation is measured with all overheads included, such as kernel launch overheads and data transfers between the host system and the GPU.

7.5.3 Performance Dependence on Number of Work-items

As discussed in chapter 5, GPUs require a large number of work-items to properly utilize the execution hardware and to hide memory latencies. It is therefore interesting to examine how many work-items we need to get decent performance.

The results are presented in figure 7.9. This is measured for a BEC with N=50 particles, but should be quite general for all the systems we study. We see that there is a sharp increase in performance until we hit 5120 work-items. We can get an idea of why this is so.

The GPU in question here is the AMD Cypress GPU, which has 20 compute units, each with 16 processing elements. In chapter 5 the Cayman GPU is described, but the only difference between Cypress and Cayman on this level is the different number of compute units, and that while Cayman's processing elements use a four-wide VLIW (Very Long Instruction Word) architecture, Cypress has a five-wide architecture.

As mentioned in chapter 5 each compute unit needs a work-group size of 64 work-items, and we arrive at 5120 work-items if we allow four such work-groups per compute unit ($20 \times 64 \times 4 = 5120$). This is apparently the minimum we require to properly hide memory latencies.

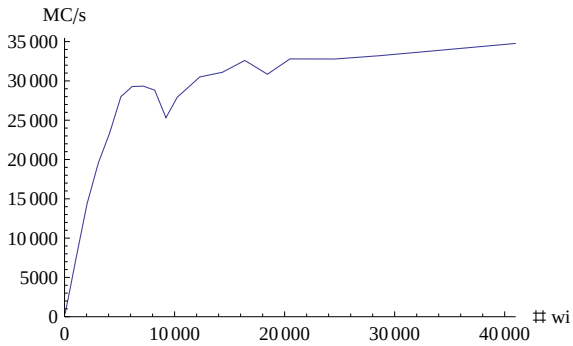


Figure 7.9: Performance in Monte Carlo cycles per second (MC/s) as a function of the number of work-items on an AMD Cypress GPU.

The performance does increase somewhat with even more work-items. I have however decided to use 5120 work-items in the performance measurements, even though the GPU is capable of about 20 % higher performance if we use tens of thousands of work-items.

It should also be noted that the work-group size is set at 64. Experimentation has shown that there is virtually no difference if it is increased to for example 128 or 256. This is not surprising since the work-group size is mostly important if local memory is used (as long as it is a multiple of the preferred size for the relevant hardware).

A nice feature is that this configuration (5120 work-items with work-group size 64) also seems to be quite optimal for the CPU. The CPU can better handle fewer work-items without losing performance, but for simplicity I have decided to use this configuration when measuring performance on the CPU as well since I haven't found a better configuration.

7.5.4 Effects of the Memory Layout

In section 6.6.4 we introduced the *arraylinear* setting, where disabling it should provide the best performance on GPUs. To study this we turn again to the BEC case with $N = 50$ particles, but the result should again be quite general.

With the *arraylinear* setting turned on the performance is 19.3 kMC/s, while with it off we get 27.7 kMC/s, an improvement of 44 %. Those results are for double precision (64-bit), where the larger data types somewhat mitigate the effects of poor memory access patterns. Using single precision floats the corresponding results are 22.0 kMC/s versus 48.8 kMC/s, an improvement of 122 %. We can note that this is when accessing vectors (particle positions and the quantum force). Had we been working with scalars the difference would likely be even larger. This shows the importance of paying attention to memory access patterns when working on the GPU.

When run on the CPU the difference is about 30 % in favor of turning *arraylinear* on for both single and double precision. Clearly the CPU benefits more from having values stored sequentially for each work-item, though the

Implementation	Helium	Beryllium	Neon
C++ reference	230 (1)	33 (1)	2.6 (1)
OpenCL CPU	1426 (6.2)	348 (10.6)	39 (15)
OpenCL GPU	15553 (68)	4056 (122)	466 (179)

Table 7.3: Performance comparisons for the atomic case. Performance given in 10^3 Monte Carlo cycles per second, with performance relative to the single-threaded reference implementation in parentheses. Note that OpenCL CPU is running on 6 CPU cores.

memory access pattern is not as important as was the case for the GPU. Based on these results we choose to measure performance with *arraylinear* off when using the GPU, and on when using the CPU.

7.5.5 The Atoms

The reference implementation used for the atomic case is an example code provided for the course FYS4411 (at the University of Oslo) in spring 2010 called fullcase.cpp. My own similar C++ implementation early in that course achieved comparable performance, therefore I believe this to be a good code for comparison[2]. The performance results are presented in table 7.3.

There is an interesting development where the relative performance of the OpenCL implementation increases for the larger systems. Both when run on the CPU and GPU the OpenCL implementation pulls ahead of the reference. The implementations are quite similar, and I do not believe there is any difference in scaling that can account for this. The more likely explanation is that there is too little computation going on for the smaller atoms (i.e. the compute to memory operation ratio is too low). Profiling information supports this. The reference implementation has the advantage of running just a single walker, meaning all data easily fits in cache. The OpenCL implementation on the other hand is spending most of its time moving data. This hits the GPU especially hard.

Note that the reference code is run single threaded, while the OpenCL CPU code is run on all 6 cores of the CPU. Therefore the reference and OpenCL CPU runs are essentially equal for helium, while OpenCL CPU is more than twice as fast for neon.

At any rate we see that the OpenCL implementation performs quite well. When run on the CPU it is competitive with the reference, and on the GPU it yields approximately two orders of magnitude higher performance.

7.5.6 The Bose Einstein Condensate

The reference implementation used for the BEC case is the implementation used in refs. [24, 25, 26]. I believe the implementation is quite similar (at least in

[2]The reason I am not using the code I developed in FYS4411 is that I switched to OpenCL in that course before finishing said implementation.

Impl.	Reference	CPU NoOpt	CPU	GPU NoOpt	GPU
Perf.	0.67 (1)	12.6 (19)	25.9 (39)	61 (91)	121 (181)

Table 7.4: Performance comparisons for the BEC case with $N = 20$ particles. Performance given in 10^3 Monte Carlo cycles per second, with performance relative to single threaded reference in parentheses. NoOpt means that the optimizations described in sections 6.4.2 and 6.5.2 are disabled (i.e. the corresponding brute force implementations are used instead). Note that the (OpenCL) CPU case is running on 6 cores.

performance characteristics) to what I might have developed in C++.

Performance numbers for this implementation can be found in ref. [26]. The CPU used there is an AMD Opteron 2218 (which is a 2.6 GHz dual-core processor anno 2006). We should therefore keep in mind that contemporary processors are faster, however today's AMD CPUs are still based on a similar architecture. It would likely be very generous to assume that the per-core performance has doubled since then.

It is stated that the serial case (i.e. single-threaded) of a BEC with $N = 20$ particles took 209 minutes moving an initial population of 4800 walkers in 1750 time steps. It is not clear how much the walker population changed, but presumably it was kept reasonably constant. This then turns out to be $4800 \times 1750 = 8.4 \times 10^6$ Monte Carlo cycles (evaluations of the local energy). With the given run-time of 209 minutes we find that the performance in our chosen measure of Monte Carlo cycles per second (MC/s) is $8.4 \times 10^6\,\mathrm{MC}/(209 \times 60\,\mathrm{s}) = 670\,\mathrm{MC/s}$.

When comparing the performance in this case there are a few complicating factors. Firstly for such a small system as this the overheads associated with transferring data to and from the GPU, along with shifting that data (which uses OpenCL vector types) to a temporary C++ array for transfer with MPI takes up about half of the total execution time. These overheads become negligible for large systems (or for the slower CPU). For example they account for about 10 % of the execution time when $N = 100$. We must therefore be aware that the OpenCL GPU case would likely look almost twice as good for larger systems than it does here.

It must also be noted that the implementation we are comparing with does not use the local energy optimization described in section 6.5.2, nor do I believe that the quantum force optimization discussed in section 6.4.2 is implemented either. Therefore I test my implementation both with and without these optimizations to get a fairer comparison.

The results are given in table 7.4. We see that even with the mentioned optimizations disabled the OpenCL implementation when run on the CPU is considerably faster than the reference CPU implementation, keeping in mind that the reference is single-threaded while the OpenCL implementation is running on 6 cores. The per-core lead is roughly three times in favor of the OpenCL implementation in this case. We must however keep in mind that the reference

implementation is running on a slower CPU (2.6 GHz versus 3.2 GHz) which is also based on an older , albeit similar architecture.

Looking at the difference between OpenCL CPU and GPU runs (with optimizations on) the GPU performs 4.7 times better than the 6 core CPU (i.e. roughly like 28 cores). Remember that the GPU is hobbled by the previously mentioned overheads in this case. Increasing to $N = 100$ particles the CPU performance is 524 MC/s while the GPU achieves 4200 MC/s, 8 times as fast (i.e. roughly like 48 CPU cores).

7.5.7 Direct Performance Analysis: FLOPS and Bandwidth

We have now looked at how the implementation performs compared to other implementations, but it is also interesting to investigate the performance in terms of how well we are utilizing the underlying hardware. The most common measures in this regard are FLOPS (floating point operations per second) and bandwidth, i.e. the rate at which data can be moved to and from the device's RAM.

We will here perform a simple analysis with a basis in the most demanding part of the code. It is however important to note that this will not yield exact figures, they will merely be (hopefully) decent approximations.

For the GPU used in this thesis (AMD Cypress) the maximum floating point performance is 2720 GFLOPS in single precision and 544 GFLOPS in double precision, and the global memory (RAM) bandwidth is 154 GB/s. These values are given in table D.4 in ref. [11].

We will be analyzing the BEC case, and the only part of the algorithm with the dubious honor of having $O\left(N^3\right)$ time complexity is the sum that was optimized in section 6.5.2. The full implementation is listed in section 6.6.6 as the function Del2WfCorrelatedAllPart. We will simplify things by only looking at the inner loop, since that should be quite dominating for large systems. The loop in question is implemented as:

```
for(uint k=j+1; k<NUMPART; k++){
  rk = r[R_IDX(k)];
  r_ik = distance(ri, rk);
  r_jk = distance(rj, rk);
  del2WfResult += 2 * (varParams.z * varParams.z
    * dot(ri - rj, ri - rk) / (r_ij * r_ij
    * r_ik * r_ik * (r_ij - varParams.z)
    * (r_ik - varParams.z)));

  //Now switch so that j and i switches places.
  del2WfResult += 2 * (varParams.z * varParams.z
    * dot(rj - ri, rj - rk) / (r_ij * r_ij
    * r_jk * r_jk * (r_ij - varParams.z)
    * (r_jk - varParams.z)));
```

```
  //And now k and i switches places instead
  del2WfResult += 2 * (varParams.z * varParams.z
    * dot(rk - rj, rk - ri) / (r_jk * r_jk
    * r_ik * r_ik * (r_jk - varParams.z)
    * (r_ik - varParams.z)));
}
```

It is the inner part of a sum of the form

$$\sum_{i=1}^{N} \sum_{j=i+1}^{N} \sum_{k=j+1}^{N} 1,$$

That sum is equivalent to the expression $\frac{1}{6}\left(2N - 3N^2 + N^3\right)$, which is the number of times the expressions in the inner loop are invoked. Now our task is to count the number of floating point operations performed by these expressions.

Firstly we have two calls to the distance built-in function of OpenCL. The distance is computed as

$$\text{distance}\left(\mathbf{r}_1, \mathbf{r}_2\right) = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2 + \left(z_1 - z_2\right)^2},$$

I count 8 basic operations (additions and multiplications) and one square root.

We also have calls to the dot built-in function, which is computed as

$$\text{dot}\left(\mathbf{r}_1, \mathbf{r}_2\right) = x_1 x_2 + y_1 y_2 + z_1 z_2,$$

I count 5 basic operations. Note that in the arguments to the dot function we are performing vector subtractions, adding another 6 operations.

We then have three similar expressions, where each one uses a single dot function (5 operations), 6 operations in the arguments to the dot function, 11 more basic operations and one division. Summing it all up I count:

- 82 basic operations (additions and multiplications)
- 2 square roots
- 3 divisions

for each time the loop body is executed. However there is still the problem of how to count the square root and division operations. Unfortunately I have been unable to find a good estimate, and so I cannot give a precise figure. I therefore choose to count them as one basic operation each, with the understanding that this likely leads to a significant underestimation of the calculated FLOPS.

For a run with $N = 100$ particles the execution time of the entire energy kernel (listed in section 6.6.7) is 798 ms for 5120 work-items. Since the system is rather large we assume that the inner loop of the Del2WfCorrelatedAllPart function accounts for most of this time, and that the number of times the loop body is executed per work-item is given by

$$\frac{1}{6}\left(2 \times 100 - 3 \times 100^2 + 100^3\right) = 161700,$$

The FLOPS estimate is then

$$5120\,\text{workitems} \times 87\,\text{operations} \times 161700\,\text{iterations}/798\,\text{ms} = 90\,\text{GFLOPS}.$$

Compared to the theoretical double precision maximum of 544 GFLOPS this may seem a bit low, but remember that it is an underestimate since there are some overheads for the rest of the energy kernel, and we counted square roots and divisions as simple operations. For example if we estimate 15 basic operations per square root and four per division the result is 129 GFLOPS. Also keep in mind that the theoretical maximum is based on the fact that multiply-accumulate operations (i.e. $a = a + b \times c$) can be done in a single instruction. For other operations the achievable rate is halved to 272 GFLOPS.

We can do a similar analysis for the bandwidth, noting that the inner loop reads a single double3 vector, which is 24 bytes. The bandwidth estimate is then

$$5120\,\text{workitems} \times 24\,\text{B} \times 161700\,\text{iterations}/798\,\text{ms} = 25\,\text{GB/s},$$

In reality a double3 is stored as a double4, which means that we could have easily gotten 33 GB/s if using the double4 type instead.

While the bandwidth is also an underestimate because of other overheads in the energy kernel, it seems likely that we are compute bound in this case. Overall I consider the results to be acceptable, as achieving the theoretical maximum performance on any architecture can be quite difficult.

7.5.8 Effects of Floating Point Precision

Unless otherwise noted all results have been with both the CPU and GPU performing computations in double precision (64-bit). Since GPUs have traditionally been geared more towards single precision computations it is interesting to examine how much of a difference there is between the two, both in terms of the results of simulations and performance.

The GPU used in this thesis can perform double precision computations at either 2/5 (for addition) or 1/5 (for everything else) speed compared to single precision. However recall from chapter 5 that AMD's GPUs use a VLIW design for their processing elements, which means that the compiler must extract ILP (Instruction Level Parallelism) from our kernels. This changes for double precision. In essence the VLIW processor operates like a scalar processor when doing double precision computations (except that it can perform two addition operations in parallel). This means that we can expect a higher utilization of the execution hardware, so that the 1/5 performance figure may be a bit misleading in real life.

It is of course also the case that when using double precision variables we have effectively halved the memory bandwidth (in terms of the number of values we can transfer). We also use twice as many registers for our floating point variables, and as mentioned in chapter 5 registers are a scarce resource.

Data type	Data cycles	Performance (kMC/s)	Result	Error
float	51.2 M	307	5.377177	0.0001
double	51.2 M	200	5.377093	0.0001

Table 7.5: Comparison of float (32-bit) and double (64-bit) computations on the GPU. System is BEC with $N = 20$, $a = 100a_{\mathrm{Rb}}$ and $\lambda = \sqrt{8}$. Result is energy per particle in units of $\hbar\omega_{\perp}$.

We saw in section 7.5.4 that double precision values are somewhat better at hiding poor memory access patterns, so we could perhaps expect a slightly higher utilization of the memory controllers when using doubles, but since we have already taken some care to use a suitable memory layout this is unlikely to amount to much.

Table 7.5 lists the results. A reasonably long run is done with both floats and doubles. In both cases the seeds to the random number generators are the same. However it is overly simplistic to say that the double result is a reference value, and that any deviation by the float run is an error. The fact is that the walks we are performing will necessarily be different when the numerical precision used to represent the particle positions differs. Therefore the energies will be slightly different, and we cannot simply say that one is more correct than the other.

A more reasonable approach is to see if the values agree within the estimated errors. In this case they do, and so it appears that floats provide sufficient precision, at least for this example computation. Since the performance is 50 % higher with floats versus doubles it seems like it would be worthwhile to use floats, but on the other hand the performance decrease may be worth it for the peace of mind the superior precision of doubles give.

Chapter 8

Conclusion

I have in this thesis developed and tested a VMC (Variational Monte Carlo) and DMC (Diffusion Monte Carlo) implementation using a combination of the C++ programming language and the OpenCL framework for parallelization, along with MPI (Message Passing Interface) for further parallelization across multiple OpenCL devices in for example a cluster. The basis in physics and statistics needed to implement VMC and DMC for atoms and BEC (Bose-Einstein Condensates) has briefly been presented, and the implementation has been described.

The experiment with OpenCL has proven successful, with the implementation achieving between one and two orders of magnitude higher performance than comparable single-threaded C/C++ implementations when run on the GPU. We have also seen that the code achieves acceptable efficiency on the GPU. The results generated agree well with values taken from other works [13, 24, 29], and by using just a single GPU we can generate results for rather large systems. It is also noteworthy that the OpenCL implementation is quite competitive when run on the CPU as well.

To the best of my knowledge DMC simulations of BEC have not previously been performed using GPGPU. The speedup achieved here indicates that larger systems could be simulated if a similar implementation was run on the new supercomputing clusters utilizing GPUs. The code could however be more suitable to large clusters, which is further discussed in the performance improvements discussion below.

The task of implementing these solvers in OpenCL did not prove much more difficult than the same task in C/C++. With the approach I chose where each work-item is an independent system the code is very similar to that of a plain C-style program, though handling the OpenCL portions of course adds some extra work. Furthermore the currently limited amounts of available standard code can add some work, like the lack of a suitable pseudo-random number generator did in my case.

The implementation I have developed of the RANLUX pseudo-random number generator (available through ref. 6.1.1) is, as far as I know, unique. The

generator's appealing qualities, like the underlying theory for its quality and the ability to select a speed/quality tradeoff, make it a potentially useful piece of software in any Monte Carlo application utilizing the OpenCL framework.

There are several possible improvements to the work I have done, outlined below.

Code Structure Improvements

While the code is designed to be extendable, it could use more work in this regard, especially for the host code. It should be much clearer how to add different systems. Furthermore I would have liked to move all physics parts out of the MonteCL class, leaving it as a pure abstraction for OpenCL with the basic building blocks for any Metropolis Monte Carlo method of this type.

Many Monte Carlo simulations have similar structures to the ones presented here for the atomic and BEC cases. Therefore a further polished version of this implementation where defining other Hamiltonians and wave functions was easier could be used to implement other simulations as well with little extra work required.

Physics and Features Improvements

Larger atoms would be interesting, for example along the lines of Sandsdalen in ref. [29]. Furthermore more properties of the BEC could be studied, such as vortexes [24].

It would also be interesting to implement the DMC method for fermionic systems, by using the fixed node approximation (see chapter 12.3.4 in ref. [18]).

Performance Improvements

There are likely more optimizations possible, for example both the local energy and the Slater determinant parts for the atomic case have been implemented in a straight-forward manner and could perhaps be somewhat optimized (though the time complexity would likely stay the same).

For the BEC case it is unfortunate that we are forced to use thousands of work-items (and thus parallel systems) per GPU to achieve acceptable performance. While running many parallel systems is generally a good thing for the quality of the result, as we increase the number of particles we will reach a point where each Monte Carlo cycle takes too long (we could also run out of memory). For a large cluster we would get fantastic statistics with hundreds of thousands of work-items, but at some point each cycle would simply take too much time. Therefore it would be useful to have more flexibility in choosing the number of parallel systems.

To this end it may be possible to have one system per work-group instead of one for each work-item. As the system sizes grow the energy kernel completely dominates the execution time, and the $O\left(N^3\right)$ computation described in section 6.5.2 could perhaps be efficiently partitioned to the work-items in a work-group.

This would allow us to reduce the needed number of parallel systems per GPU from thousands to less than one hundred, giving us much more flexibility. Indeed if local memory could be utilized cleverly such an approach could potentially lead to a further performance increase of the energy calculation as well.

Bibliography

[1] *FYS4411 Lecture Note Slides, University of Oslo, Autumn 2010.*

[2] *http://cpc.cs.qub.ac.uk/.*

[3] *http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx.*

[4] *http://irs.inms.nrc.ca/software/egsnrc-V4-2.3.1/.*

[5] *http://luscher.web.cern.ch/luscher/ranlux/index.html.*

[6] *https://bitbucket.org/ivarun/montecl.*

[7] *https://bitbucket.org/ivarun/ranluxcl.*

[8] *http://www.honeylocust.com/RngPack/.*

[9] *http://www.khronos.org/registry/cl/.*

[10] *http://www.pgroup.com/resources/cuda-x86.htm.*

[11] *AMD Accelerated Parallel Processing Programming Guide.* May 2011.

[12] B. L. Hammond, W. A. Lester, Jr., and P. J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry.* World Scientific, 1994.

[13] D Blume and C H Greene. Quantum corrections to the ground state energy of a trapped Bose-Einstein condensate: A diffusion Monte Carlo calculation. 2000. arXiv:cond-mat/0009220v2.

[14] Franco Dalfovo, Stefano Giorgini, Lev P. Pitaevskii, and Sandro Stringari. Theory of Bose-Einstein condensation in trapped gases. *Reviews of Modern Physics*, 71:463, 1999.

[15] F. James. A review of pseudorandom number generators. *Computer Physics Communications*, 60:329, 1990.

[16] F. James. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, 79:111, 1994.

[17] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29:610, 1958.

[18] J. M. Thijssen. *Computational Physics*. Cambridge University Press, 2007.

[19] W. Krauth. *Statistical Mechanics: Algorithms and Computations*. Oxford University Press, 2006.

[20] Lev N. Shchur and Paolo Butera. The RANLUX generator: resonances in a random walk test. *International Journal of Modern Physics C*, 9:607, 1998.

[21] M. H. Anderson, J. R. Ensher, and M. R. Matthews. Observation of Bose-Einstein Condensation in a Dilute Atomic Vapor. *Science*, 269:198, 1995.

[22] Martin Lüscher. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, 79:100–110, 1994.

[23] Morten Hjorth-Jensen. *Computational Physics*. University of Oslo, 2010. FYS3150 lecture notes.

[24] J. K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, and A. Polls. Vortices in atomic Bose-Einstein condensates in the large-gas-parameter region. *Physical Review A*, page 053610, 2005.

[25] Jon Kristian Nilsen. Bose-Einstein condensation in trapped bosons: A quantum Monte Carlo analysis. Master's thesis, University of Oslo, 2004.

[26] Jon Kristian Nilsen. MontePython: Implementing Quantum Monte Carlo using Python. *Computer Physics Communications*, 177:799, 2007.

[27] R. Shankar. *Principles of Quantum Mechanics 2nd ed.* Springer, 1994.

[28] A. R. Sakhel, J. L. DuBois, and H. R. Glyde. Bose-Einstein condensates in ^{85}Rb gases at higher densities. *Physical Review A*, 66:063610, 2002.

[29] Håvard Sandsdalen. Variational Monte Carlo studies of atoms. Master's thesis, University of Oslo, 2010.

[30] Vadim Demchik. Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units. *Computer Physics Communications*, 182:692, 2011.

[31] W. H. Press, S. A. Teukolsky, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.