ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Implementation of the two-dimensional quantum lattice Boltzmann scheme in CUDA

Bachelor Thesis

Fabian Thüring

June, 2015

Advisors: Prof. Dr. H. J. Herrmann, Dr. M. Mendoza

Department of Physics, ETH Zürich

## Abstract

The quantum lattice Boltzmann (QLB) scheme has shown to be a viable computational approach to simulate quantum systems described by the Schrödinger or Dirac equation, hence providing a unified treatement of relativistic and non-relativistic quantum mechanics. Among others, the QLB attempt allows an efficient and scalable implementation on modern parallel computer architectures.

The goal of this bachelor thesis is to provide a tool kit to study and solve the Dirac equation in two spatial dimensions with the quantum lattice Boltzmann scheme. The resulting implementation allows real-time visualization of various aspects of the solution by means of GPU acceleration using the compute unified device architecture (CUDA) as a software development platform.

# Contents

Chapter 1

# Introduction

Since the early days of quantum theory, it has been pointed out that there are some formal analogy between fluid mechanics and quantum mechanics. This has lead to the development of numerical methods exporting assets from computational fluid dynamics to the quantum world. The method of interest in this thesis will be the quantum lattice Boltzmann (QLB) approach. It is built upon a formal analogy between the discrete kinetic Boltzmann equation and the Dirac equation. The Dirac equation is one of the most fundamental equation in modern physics, as it offers a description of spin-½ particles which is compatible with the theory of special relativity.

## 1.1 Motivation

The Dirac equation has a wide field of application. For example, the description of electron transport in graphene, a single atomic layer of carbon, is modelled by the two-dimensional Dirac equation, as the charge carriers in graphene can be represented by relativistic particles with zero rest mass and an effective speed of light of $c \approx 10^{-6}\, ms^{-1}$ [2]. Further applications may be found in simulating Einstein-Bose condensates which are described by a non-linear Dirac equation [3]. In this regard, we are going to solve the Dirac equation in two spatial dimensions by means of the QLB approach [4]. This thesis will be dedicated to provide an extensive tool kit[1] to study the Dirac equation on the computer. Which will range from simple input libraries, to create initial conditions, over to a high performance implementation of the QLB scheme.

Besides the Dirac solver, a 3D visualization framework will be developed that allows to display the various aspects of the solution in real-time. This

[1]Code available under https://github.com/thfabian/QLB

will enable the user of the program to immediately get a visual feedback and hopefully help to get a better grasp of the simulated physics.

As these set goals are computationally demanding, we have to exploit the parallelism in modern multi- and many-core processors. The latter will be achieved by using Nvidia's Compute Unified Device Architecture (CUDA) which provides a software development platform to run parts of the program on the graphics processing unit. A major part of the thesis will be spent in discussing strategies of efficiently implementing the QLB scheme in parallel in a shared memory environment.

## 1.2 Structure of the thesis

The remainder of this thesis is structured as follows:

- **Chapter 2** gives a short recap of quantum theory needed for this thesis and introduces some common notation.
- **Chapter 3** explains the one- and two-dimensional QLB scheme in detail.
- **Chapter 4** discusses the implementation details of the parallel CPU as well as the CUDA version in depth.
- **Chapter 5** illustrates the validation of the numerical scheme by doing some simple calculation.
- **Chapter 6** describes the usage of the code and showcases an actual example.
- **Chapter 7** presents the conclusion of this thesis and outlines some possible extensions for future work.

Chapter 2

# Quantum Theory

This chapter will shortly recap the basics of quantum theory needed for this thesis. First, we are going to take a closer look at the Dirac equation and latter investigate a formal analogy between the Dirac equation and the discrete-velocity version of the Boltzmann equation.

## 2.1 Dirac equation

The Dirac equation, derived by Paul Dirac in 1928, allows a quantum mechanical description of an electron that meets the requirements of special relativity. In standard form the Dirac equation reads as follows [4]:

$$(\partial_t + c\,\alpha^x\partial_x + c\,\alpha^y\partial_y + c\,\alpha^z\partial_z)\,\psi = (-iw_c\beta + igI)\psi \tag{2.1}$$

with $c$ being the speed of light, $\hbar$ the reduced Planck's constant, $w_c = mc^2/\hbar$ the Compton frequency of a particle of mass $m$ and $g = qV/\hbar$ couples the wave function to a scalar potential $V$ with $q$ being the electric charge.

The wave function $\psi = \psi(\vec{x}, t)$ describes a complex four dimensional spinor (*4-spinor*) which can be interpreted as a particle-antiparticle pair of an electron. The $4 \times 4$ hermitian matrices $\alpha^x, \alpha^y, \alpha^z, \beta$ are commonly referred to as Dirac matrices [5] and are given in terms of the $2 \times 2$ Pauli matrices $\sigma^k, \quad k = x, y, z$:

$$\alpha^k = \begin{pmatrix} 0 & \sigma^k \\ \sigma^k & 0 \end{pmatrix} \qquad \beta = \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix}$$

where $I$ is the $2 \times 2$ identity matrix, with

$$\sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma^z = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

leading to:

$$\alpha^x = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \alpha^y = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

$$\alpha^z = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \qquad \beta = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

It is of utmost importance that Pauli spin matrices do not commute, meaning there is no basis in which all 3 Pauli matrices are diagonal. The same holds for the Dirac matrices build on top of them.

For convenience, we will work with natural units ($\hbar = c = 1$) for the rest of this thesis. Henceforth, we work with the following representation of the Dirac equation:

$$(\partial_t + \alpha^x \partial_x + \alpha^y \partial_y + \alpha^z \partial_z)\, \psi = (-im\beta + igI)\psi \tag{2.2}$$

## 2.2 Disrecte Boltzmann and Dirac

As mentioned in the previous chapter, the formal analogy between fluid and quantum mechanics has been known for a long time. It seems therefore natural to ask if we can adopt schemes from computational fluid dynamics to be used to model evolutionary quantum mechanical systems. The approach we are going to investigate further has been developed in the last decades [4, 7, 9] by *Succi et al.* It is based upon exporting assets from the lattice Boltzmann equation to the quantum world.

### 2.2.1 Discrete Boltzmann equation

The lattice Boltzmann approach as described in [9] comes in very handy when solving problems in computational fluid dynamics. These advantages stem from the fact that one does not try to solve the Navier-Stokes equations directly, instead recovers them as equations that describe slowly varying solutions of the discrete Boltzmann equation:

$$(\partial_t + v_i^x \partial_x + v_i^y \partial_y + v_i^z \partial_z)\, f_i = \sum_{j=0}^{n} \Omega_{ij}(f_j - f_j^{eq}) \tag{2.3}$$

for $i = 0, ..., n$ with n representing the directions of the lattice and $f_i = f(\vec{x}, \vec{v} = \vec{v}_i, t)$ being the probability density of finding a particle at time $t$ around position $\vec{x}$ with discrete velocity $\vec{v}_i = (v_i^x, v_i^y, v_i^z)$. The left hand side of the equation can be seen as the free-streaming, while the right hand side, the collision part, drives the probability density towards a local Maxwell equilibrium $f_j^{eq}$. The collision matrix $\Omega_{ij}$ encodes mass, momentum and energy conservation [6].

The Boltzmann equation (2.3) has several useful features: it is a linear, constant coefficient, hyperbolic system. When discretized on a lattice we can integrate along characteristics which yields the *lattice* Boltzmann equation [10]:

$$f_i(\vec{x} + \Delta\vec{x}, t + \Delta t) - f_i(\vec{x}, t) = \sum_{j=0}^{n} \Omega_{ij}(f_j - f_j^{eq}) \tag{2.4}$$

Another very useful feature is that all non-linearity is restricted to algebraic terms that are local to each grid point and are confined in the collision matrix.

### 2.2.2 Comparison to the Dirac equation

It has been shown [9] that the same approach, concerning the Navier-Stokes and Boltzmann equation, can also be applied to the Schrödinger and Dirac equation. We are not solving the Schrödinger equation directly, instead recover it as the limiting equation that describes slowly varying solutions of the Dirac equation. If we recall the Dirac equation (2.2):

$$(\partial_t + \alpha^x\partial_x + \alpha^y\partial_y + \alpha^z\partial_z)\,\psi = (-im\beta + igI)\psi$$

The structural similarity to Eq. (2.3) is intriguing. Both equations only contain first order derivatives in space and time. The distribution functions $f_i$ are analogous to the four components of the wave function $\psi_i$. In addition, the set of discrete velocities $\vec{v}_i$ is analogous to the set of Dirac matrices $\alpha^x, \alpha^y, \alpha^z$ as pointed out in [4]. The collision matrix $\Omega_{ij}$ can be related to the matrix $\beta$ in the Dirac equation. In fact, in 1D the analogy is exact as we can choose a basis where the streaming matrix is diagonal.

In higher dimension however, there is some mismatch in degrees of freedom: A spinor of order $s$ will always contain $2s + 1$ components, independent of the dimension, while the discrete distribution functions $f_i$ need atleast $2d$ components depending on the grid ($d$ representing the dimensions). As already mentioned in the last section, a structural mismatch occurs due to the impossibility to simultaneously diagonalize the Dirac matrices $\alpha^x, \alpha^y, \alpha^z$. To tackle these problems we have to apply a technique called operator splitting, which will be discussed in 3.2.

## 2.3 Macroscopic variables

In order to extract some further information out of the Dirac equation, likewise the probability density and probability current, we have to find an equation of continuity for the probability. To do so, we reorder the Dirac equation (2.2):

$$\partial_t\psi + \alpha^x\partial_x\psi + \alpha^y\partial_y\psi + \alpha^z\partial_z\psi + im\beta\psi - ig\psi = 0 \tag{2.5}$$

and multiply the whole equation with $\beta$ and set $g \equiv 0$:

$$\beta\partial_t\psi + \beta\alpha^x\partial_x\psi + \beta\alpha^y\partial_y\psi + \beta\alpha^z\partial_z\psi + im\psi = 0 \tag{2.6}$$

For convenience, we define a new set of matrices:

$$\gamma^0 = \beta, \quad \gamma^1 = \beta\alpha^x, \quad \gamma^2 = \beta\alpha^y, \quad \gamma^3 = \beta\alpha^z \tag{2.7}$$

with the property:

$$\gamma^0\gamma^0 = I \tag{2.8}$$

Next, we plug in the newly defined matrices and take the Hermitian conjugate of (2.6):

$$[\gamma^0\partial_t\psi + \gamma^1\partial_x\psi + \gamma^2\partial_y\psi + \gamma^3\partial_z\psi + im\psi]^\dagger = 0 \tag{2.9}$$

hence,

$$\partial_t\psi^\dagger\gamma^{0\dagger} + \partial_x\psi^\dagger\gamma^{1\dagger} + \partial_y\psi^\dagger\gamma^{2\dagger} + \partial_z\psi^\dagger\gamma^{3\dagger} - im\psi^\dagger = 0 \tag{2.10}$$

By using an easy-to-verify anti-hermitian relation of the newly defined matrices:

$$\gamma^{k\dagger} = -\gamma^k \qquad k = 1, 2, 3 \tag{2.11}$$

and $\gamma^{0\dagger} = \gamma^0$ we can rewrite equation (2.10) as:

$$\partial_t\psi^\dagger\gamma^0 + \partial_x\psi^\dagger(-\gamma^1) + \partial_y\psi^\dagger(-\gamma^2) + \partial_z\psi^\dagger(-\gamma^3) - im\psi^\dagger = 0 \tag{2.12}$$

We would like to write (2.12) in a covariant fashion, meaning $\psi^\dagger(\partial_k\gamma^k - im)$ where $\partial_k$ acts on the left. On the first glance, this seems not possible due to the minus sign in front of the spatial components of the $\gamma^k$ matrices. Luckily, we can flip the sign on the $\gamma^k$ matrices by multiplying them with $\gamma^0$ from the right i.e $-\gamma^k\gamma^0 = \gamma^0\gamma^k$. This yields:

$$\partial_t\psi^\dagger\gamma^0\gamma^0 + \partial_x\psi^\dagger(-\gamma^1\gamma^0) + \partial_y\psi^\dagger(-\gamma^2\gamma^0) + \partial_z\psi^\dagger(-\gamma^3\gamma^0) - im\psi^\dagger\gamma^0\gamma^0 = 0 \tag{2.13}$$

and

$$\partial_t\psi^\dagger\gamma^0\gamma^0 + \partial_x\psi^\dagger(\gamma^0\gamma^1) + \partial_y\psi^\dagger(\gamma^0\gamma^2) + \partial_z\psi^\dagger(\gamma^0\gamma^3) - im\psi^\dagger\gamma^0\gamma^0 = 0 \tag{2.14}$$

by defining the adjoint spinor $\overline{\psi} = \psi^\dagger \gamma^0$ and using (2.8), we can define the *adjoint Dirac equation*:

$$\partial_t \overline{\psi} \gamma^0 + \partial_x \overline{\psi} \gamma^1 + \partial_y \overline{\psi} \gamma^2 + \partial_z \overline{\psi} \gamma^3 - im\overline{\psi} = 0 \tag{2.15}$$

or in a more compact notation:

$$\overline{\psi}(\partial_k \gamma^k - im) = 0 \tag{2.16}$$

Similarly, we can write the Dirac equation (2.2) (taking $g \equiv 0$) as:

$$(\gamma^k \partial_k + im)\psi = 0 \tag{2.17}$$

Now, let us multiply the Dirac equation (2.17) with $\overline{\psi}$ from the left:

$$\overline{\psi}(\gamma^k \partial_k + im)\psi = 0 \tag{2.18}$$

Consequently, we multiply the adjoint Dirac equation (2.16) with $\psi$ from the right:

$$\overline{\psi}(\partial_k \gamma^k - im)\psi = 0 \tag{2.19}$$

If we add up (2.18) and (2.19) the mass term drops out, leading to:

$$\overline{\psi}(\gamma^k \partial_k)\psi + \overline{\psi}(\partial_k \gamma^k)\psi = 0 \tag{2.20}$$

which is equivalent to:

$$\partial_k(\overline{\psi}\gamma^k \psi) = 0 \tag{2.21}$$

and thus by replacing the $\gamma^k$ matrices (2.7) with the Dirac matrices $\alpha^k$ and using (2.8) and $\overline{\psi} = \psi^\dagger \gamma^0 = \psi^\dagger \beta$ we finally get our desired equation of continuity:

$$\partial_t(\underbrace{\overline{\psi}\gamma^0\psi}_{\rho}) + \partial_x(\underbrace{\overline{\psi}\gamma^1\psi}_{j_x}) + \partial_y(\underbrace{\overline{\psi}\gamma^2\psi}_{j_y}) + \partial_z(\underbrace{\overline{\psi}\gamma^3\psi}_{j_z}) = 0 \tag{2.22}$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \vec{j} = 0 \tag{2.23}$$

with $\rho$ being the probability density and $\vec{j}$ the probability current:

$$\rho = \psi^\dagger \gamma^0 \gamma^0 \psi = \psi^\dagger \psi \tag{2.24}$$

$$j_x = \psi^\dagger \gamma^0 \gamma^1 \psi = \psi^\dagger \alpha^x \psi \tag{2.25}$$

$$j_y = \psi^\dagger \gamma^0 \gamma^2 \psi = \psi^\dagger \alpha^y \psi \tag{2.26}$$

$$j_z = \psi^\dagger \gamma^0 \gamma^3 \psi = \psi^\dagger \alpha^z \psi \tag{2.27}$$

In addition, we can define the probability velocity:

$$v_x = \frac{j_x}{\rho} \tag{2.28}$$

$$v_y = \frac{j_y}{\rho} \tag{2.29}$$

$$v_z = \frac{j_z}{\rho} \tag{2.30}$$

These quantities will be of great use when visualizing the solution in the simulation as presented in section 4.2.

Chapter 3

# Quantum Lattice Boltzmann

As pointed out in the last chapter, the Dirac equation (2.2) has some astonishing structural similarities to the lattice Boltzmann equation (2.3). Hence, we are tempted to apply the same scheme used to solve the lattice Boltzmann equation to the Dirac equation. This approach is known as the Quantum Lattice Boltzmann (QLB) scheme [4]. In this chapter we are going to derive the QLB scheme in one spatial dimension and latter extend it via operator splitting to two spatial dimensions.

## 3.1 One-dimensional quantum lattice Boltzmann scheme

The analogy to the lattice Boltzmann equation (2.3) is only valid if the Dirac matrix $\alpha^x$ is in diagonal form. The one-dimensional Dirac equation can be written as:

$$(\partial_t + \alpha^x \partial_x)\,\psi = (-im\beta + igI)\psi \tag{3.1}$$

with

$$\alpha^x = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

being the Dirac matrix introduced in 2.1. Clearly, the matrix $\alpha^x$ is not diagonal. However, $\alpha^x$ is a hermitian matrix and therefore we can always find an unitary transformation to diagonalize it. We may choose:

$$X^{-1}\alpha^x X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = \beta$$

with the set of unitary matrices:

$$X = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \tag{3.2}$$

and

$$X^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \tag{3.3}$$

We thus diagonalize $\alpha^x$ in (3.1) by multiplying with $X^{-1}$ from the left, leading to:

$$X^{-1}(\partial_t + \alpha^x \partial_x)\,\psi = X^{-1}(-im\beta + igI)\psi \tag{3.4}$$

inserting an identity matrix $I = XX^{-1}$ delivers:

$$X^{-1}(\partial_t + \alpha^x \partial_x)XX^{-1}\,\psi = X^{-1}(-im\beta + igI)XX^{-1}\psi \tag{3.5}$$

and finally,

$$(\partial_t + X^{-1}\alpha^x X\partial_x)X^{-1}\,\psi = \underbrace{X^{-1}(-im\beta + igI)X}_{\text{Q}}\,X^{-1}\psi \tag{3.6}$$

Using $X^{-1}\alpha^x X = \beta$ and defining the rotated collision matrix $Q$ as shown above, we obtain:

$$(\partial_t + \beta\partial_x)X^{-1}\,\psi = QX^{-1}\psi \tag{3.7}$$

with

$$Q = \begin{pmatrix} ig & 0 & 0 & -im \\ 0 & ig & im & 0 \\ 0 & im & ig & 0 \\ -im & 0 & 0 & ig \end{pmatrix}$$

We can rewrite (3.7) as a system of of partial differential equations. We define the rotated wave function $X^{-1}\psi = (u_1, u_2, d_1, d_2)^T$ with $u$ and $d$ propagating up and down the x axis, respectively, and the subscripts (1) and (2) indicating the spin states. The resulting system of equations is:

$$\begin{aligned} \partial_t u_1 + \partial_x u_1 &= igu_1 - imd_2 \\ \partial_t u_2 + \partial_x u_2 &= igu_2 + imd_1 \\ \partial_t d_1 - \partial_x d_1 &= igd_1 + imu_2 \\ \partial_t d_2 - \partial_x d_2 &= igd_2 - imu_1 \end{aligned} \tag{3.8}$$

**Discretization** As described in [4] and [10] the left hand side of the equations in (3.8) can be solved by integrating along the characteristics. The characteristics can be associated with the light cones $(x \pm cs, t+s)$ parametrized by $s$. Therefore, the integration yields exactly the difference between the two ends of the characteristic. The right hand side can be discretized by using a Crank-Nicolson quadrature rule, which requires the average of the variables at time $t$ and position $x$ as well as at time $t+\Delta t$ and $x \pm \Delta x$. The discretized system of equations (3.8) yields:

$$\begin{aligned}
\widehat{u_1} - u_1 &= \tfrac{1}{2} i\widetilde{g}(\widehat{u_1} + u_1) - \tfrac{1}{2} i\widetilde{m}(\widehat{d_2} + d_2)\\
\widehat{u_2} - u_2 &= \tfrac{1}{2} i\widetilde{g}(\widehat{u_2} + u_2) + \tfrac{1}{2} i\widetilde{m}(\widehat{d_1} + d_1)\\
\widehat{d_1} - d_1 &= \tfrac{1}{2} i\widetilde{g}(\widehat{d_1} + d_1) + \tfrac{1}{2} i\widetilde{m}(\widehat{u_2} + u_2)\\
\widehat{d_2} - d_2 &= \tfrac{1}{2} i\widetilde{g}(\widehat{d_2} + d_2) - \tfrac{1}{2} i\widetilde{m}(\widehat{u_1} + u_1)
\end{aligned} \tag{3.9}$$

where $(\widehat{\ })$ indicates the variable is evaluated somewhere else than $(x,t)$:

$$\begin{aligned}
\widehat{u_1} &= u_1(x+\Delta x, t+\Delta t), & u_1 &= u_1(x,t)\\
\widehat{u_2} &= u_2(x+\Delta x, t+\Delta t), & u_2 &= u_2(x,t)\\
\widehat{d_1} &= d_1(x-\Delta x, t+\Delta t), & d_1 &= d_1(x,t)\\
\widehat{d_2} &= d_2(x-\Delta x, t+\Delta t), & d_2 &= d_2(x,t)
\end{aligned}$$

with $\widetilde{g} = g(x,t)\Delta t$ as well as $\widetilde{m} = m\Delta t$. One can solve the system of equation (3.9) algebraically to obtain explicit formulas for the $(\widehat{\ })$ variables, namely:

$$\begin{aligned}
\widehat{u_1} &= au_1 - ibd_2\\
\widehat{u_2} &= au_2 + ibd_1\\
\widehat{d_1} &= ad_1 + ibu_2\\
\widehat{d_2} &= ad_2 - ibu_1
\end{aligned} \tag{3.10}$$

with the coefficients $a$ and $b$:

$$a = \frac{1 - \Omega/4}{1 + \Omega/4 - i\widetilde{g}} \qquad b = \frac{\widetilde{m}}{1 + \Omega/4 - i\widetilde{g}} \qquad \Omega = \widetilde{m}^2 - \widetilde{g}^2 \tag{3.11}$$

The right hand side of equation (3.10) is just a multiplication of the rotated wave function $X^{-1}\psi = (u_1, u_2, d_1, d_2)^T$ with the collision matrix $\widehat{Q}_X$.

$$\widehat{Q}_X = \begin{pmatrix} a & 0 & 0 & -bi \\ 0 & a & bi & 0 \\ 0 & bi & a & 0 \\ -bi & 0 & 0 & a \end{pmatrix} \tag{3.12}$$

Clearly, $\widehat{Q}_X$ is an unitary matrix which can be shown by using the fact that the coefficients satisfy the relation: $|a|^2 + |b|^2 = 1$. The overall QLB scheme is therefore nothing else than evolving the wave function with unitary operations.

**Algorithm** We have now derived all the ingredients to define the final algorithm for the one dimensional QLB scheme. To march one step in time we have to do the following procedure:

1. Rotate the *4-spinor* $\psi$ with $X^{-1}$
2. Stream and collide with $\widehat{Q}_X$
3. Rotate back with $X$

The subtle differences to the classical LB scheme (2.4) are the rotation and anti rotation steps to assure the Dirac matrix $\alpha^x$ is in diagonal form.

## 3.2 Extension to two spatial dimensions

To extend our previously derived algorithm to higher dimensions, we have to account for the matter that there is no basis in which all Dirac matrices are simultaneously diagonal. The solution to this dilemma is a technique well-known in numerical mathematics: *operator splitting*. We don't try to solve the two-dimensional Dirac equation directly, instead decompose it in a sum of two one-dimensional equations. We write the two-dimensional Dirac equation as:

$$\left(\partial_t + [\alpha^x\partial_x + \tfrac{1}{2}(im\beta - igI)] + [\alpha^y\partial_y + \tfrac{1}{2}(im\beta - igI)]\right)\psi = 0 \tag{3.13}$$

Where the brackets $[\dots]$ denote the streaming and collision operator in one spatial direction. Thus, we solve each individual equation $\partial_t\psi = [...]\psi$ separately. This leads to the following scheme for computing the evolution over a time interval $\Delta t$:

$$\psi' = exp\{\Delta t[\alpha^x\partial_x + \tfrac{1}{2}(im\beta - igI)]\}\psi(t) \tag{3.14}$$

$$\psi(t+\Delta t) = exp\{\Delta t[\alpha^y\partial_y + \tfrac{1}{2}(im\beta - igI)]\}\psi' \tag{3.15}$$

with a given initial condition $\psi(0) = \psi_0$. Clearly, this approach is not exact. The two operators do not commute and we commit an $\mathcal{O}(\Delta t^2)$ splitting error by using the product of exponentials to approximate the exponential of the sum of the two operators.

**Extended operator splitting** Simple operator splitting still won't seal the deal. The key observation here is to realize that we *do not* have to work with the same representation of the Dirac equation in each separate streaming step [9]. We are therefore free to choose a basis in which the Dirac matrix $\alpha^k$ for $k = x, y$ is diagonal. To diagonalize $\alpha^y$ we may choose the matrix pair $Y$ and $Y^{-1}$:

$$Y = \frac{1}{\sqrt{2}}\begin{pmatrix} -i & 0 & 0 & -i \\ 0 & 1 & 1 & 0 \\ 0 & -i & i & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \qquad Y^{-1} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 0 & -i \\ 0 & 1 & i & 0 \\ 0 & 1 & -i & 0 \\ i & 0 & 0 & -1 \end{pmatrix} \tag{3.16}$$

in which the unitary collision matrix $\widehat{Q}_Y$ has the following representation:

$$\widehat{Q}_Y = \begin{pmatrix} a & 0 & 0 & -bi \\ 0 & a & -bi & 0 \\ 0 & -bi & a & 0 \\ -bi & 0 & 0 & a \end{pmatrix} \tag{3.17}$$

with $a$ and $b$ defined in (3.11).

### 3.2.1 Algorithm

The final algorithm for the two-dimensional QLB scheme is just an execution of two subsequent one-dimensional schemes.

1. Rotate the *4-spinor* $\psi$ with $X^{-1}$
2. Stream and collide with $\widehat{Q}_X$
3. Rotate back with $X$
4. Rotate the *4-spinor* $\psi$ with $Y^{-1}$
5. Stream and collide with $\widehat{Q}_Y$
6. Rotate back with $Y$

With the matrices as defined in (3.2), (3.3), (3.12), (3.16) and (3.17).

$$X = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \qquad X^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$

$$Y = \frac{1}{\sqrt{2}} \begin{pmatrix} -i & 0 & 0 & -i \\ 0 & 1 & 1 & 0 \\ 0 & -i & i & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \qquad Y^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & -i \\ 0 & 1 & i & 0 \\ 0 & 1 & -i & 0 \\ i & 0 & 0 & -1 \end{pmatrix}$$

$$\widehat{Q}_X = \begin{pmatrix} \widetilde{a} & 0 & 0 & -\widetilde{b}i \\ 0 & \widetilde{a} & \widetilde{b}i & 0 \\ 0 & \widetilde{b}i & \widetilde{a} & 0 \\ -\widetilde{b}i & 0 & 0 & \widetilde{a} \end{pmatrix} \qquad \widehat{Q}_Y = \begin{pmatrix} \widetilde{a} & 0 & 0 & -\widetilde{b}i \\ 0 & \widetilde{a} & -\widetilde{b}i & 0 \\ 0 & -\widetilde{b}i & \widetilde{a} & 0 \\ -\widetilde{b}i & 0 & 0 & \widetilde{a} \end{pmatrix}$$

Note that $\widetilde{a}$ and $\widetilde{b}$ are defined as:

$$\widetilde{a} = \frac{1 - \Omega_2/4}{1 + \Omega_2/4 - i\widetilde{g}_2} \qquad \widetilde{b} = \frac{\widetilde{m}_2}{1 + \Omega_2/4 - i\widetilde{g}_2} \qquad \Omega_2 = \widetilde{m}_2^2 - \widetilde{g}_2^2$$

with $\widetilde{m}_2 = \frac{1}{2}m\Delta t$ and $\widetilde{g}_2 = \frac{1}{2}g(\vec{x}, t)\Delta t$ because we are doing a time step of $\frac{1}{2}\Delta t$ in each direction.

Chapter 4

# Implementation Details

In this chapter we will discuss the core of the thesis: the implementation of the QLB scheme in two spatial dimensions with real-time visualization of the solution. First, a closer look at the general implementation framework, including the visualization, is taken and afterwards the details of the multi-threaded CPU as well as the CUDA implementation are presented.

## 4.1 Computational aspects

The implementation is completely written in `C++` with minimal dependencies on third-party libraries. In fact, only a few OpenGL libraries are required. This enables simpler portability across different operating systems. This section will focus on the implementation of the QLB algorithm described in the last chapter. Further details on the structure of the code and usage are being presented in chapter 6.

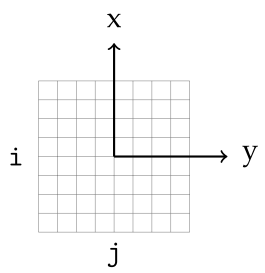
### 4.1.1 Lattice discretization

As observed in [9] the QLB scheme does *not* require a special lattice as may needed by classical LB schemes. We can therefore freely work with a simple square lattice. We discretize the domain in a sequence of $L$ points in each dimension with a *fixed* uniform mesh size of $\Delta x$. The origin $(0,0)$ will be placed at the grid point $(\frac{L-1}{2}, \frac{L-1}{2})$ leading to the relation between grid points $(i,j)$ and domain $(x_i, y_j)$ as shown in Fig. 4.1.

The covered domain is therefore given by:

$$[-\frac{\Delta x}{2}(L-1), \frac{\Delta x}{2}(L-1)] \times [-\frac{\Delta x}{2}(L-1), \frac{\Delta x}{2}(L-1)]$$

**Working variables** For each grid point $(i,j)$ we have to store a *4-spinor*. This leads to an array of complex numbers of size $4 \cdot L^2$ which we will refrence as



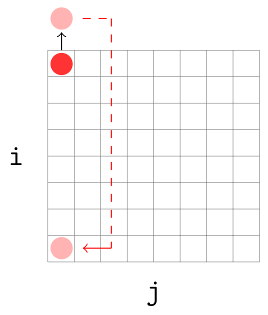
$$x_i = \Delta x(i - \tfrac{1}{2}(L-1)$$
$$y_j = \Delta x(j - \tfrac{1}{2}(L-1)$$

**Figure 4.1:** Spatial domain decomposition of a $8 \times 8$ grid

spinor. Due to the nature of the scheme, we have to allocate two additional temporary arrays, called `spinorrot` and `spinoraux`. It comes as no surprise that the total memory usage is of order $\mathcal{O}(L^2)$ or more precisely, we have to store $24 \cdot L^2$ floating point numbers in total as a complex number is usually represented by two floating point numbers.

### 4.1.2 Periodic boundary conditions

In the streaming steps (2 and 5) of the algorithm 3.2.1 we have to access the nearest neighbours, as depicted in Fig 4.2. This might be a problem if the grid cell lies on the boundary. A common solution to this is to impose periodic boundary conditions. The boundaries simply interact with the boundaries on the other side of the lattice.



This leads to the following rules to access the neighbours:

$$\begin{aligned} \texttt{ia} &= (\texttt{i + 1})\,\%\,\texttt{L} \\ \texttt{ik} &= (\texttt{i - 1 + L})\,\%\,\texttt{L} \end{aligned} \tag{4.1}$$

and

$$\begin{aligned} \texttt{ja} &= (\texttt{j + 1})\,\%\,\texttt{L} \\ \texttt{jk} &= (\texttt{j - 1 + L})\,\%\,\texttt{L} \end{aligned} \tag{4.2}$$

respectively. With % representing the modulo operation.

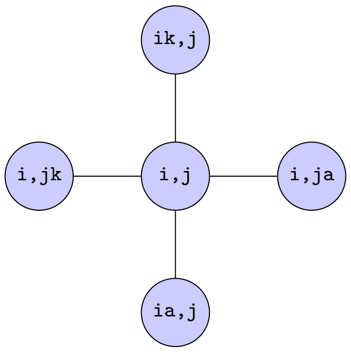


**Figure 4.2:** Stencil of the streaming step

### 4.1.3 Numerical scheme

If we incorporate everything discussed in the above sections and follow the algorithm described in 3.2.1, we have to do the following for *each* grid point $(i, j)$:

In $x$-direction:

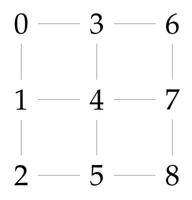
1. Multiply `spinor[i,j]` with $X^{-1}$ and store the result in `spinorrot[i,j]`.
2. Copy `spinorrot[i,j]` to `spinoraux[i,j]`.
3. Calculate $\widehat{Q}_X$ and multiply `spinoraux[i,j]` with $\widehat{Q}_X$. Store the result in `spinorrot[ia,j]` and `spinorrot[ik,j]`, respectively.
4. Multiply `spinorrot[i,j]` with $X$ and store the result in `spinor[i,j]`.

and the same in $y$-direction:

5. Multiply `spinor[i,j]` with $Y^{-1}$ and store the result in `spinorrot[i,j]`.
6. Copy `spinorrot[i,j]` to `spinoraux[i,j]`.
7. Calculate $\widehat{Q}_Y$ and multiply `spinoraux[i,j]` with $\widehat{Q}_Y$. Store the result in `spinorrot[i,ja]` and `spinorrot[i,jk]`, respectively.
8. Multiply `spinorrot[i,j]` with $Y$ and store the result in `spinor[i,j]`.

## 4.2 Visualization

As mentioned in the introduction, one of the goal of this thesis is to provide a *real-time* 3D visualization. To achieve this goal we have to use a fairly low level graphics API, like OpenGL. OpenGL is implemented as a state machine, meaning once we have set all the necessary parameters, we can simply feed-in the vertices and normals and the scene will be rendered. To begin with, we have to determine what we are actually going to display.

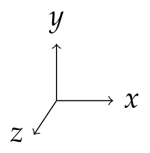


```
index = [1, 0, 3, 4, 1, 3, 2, 1, 4, 5, 2, 4, 4,
         3, 6, 7, 4, 6, 5, 4, 7, 8, 5, 7 ]
```

**Figure 4.3:** Example of an `index` array for a $3 \times 3$ grid. Three consecutive vertices will span a triangle. In this case a total of 8 triangles would be rendered.

### 4.2.1 Vertex calculation

Clearly, we cannot directly visualize the *4-spinors* as we need a real-valued scalar for each grid point. Instead, we can either visualize the probability density of the individual spinors $|\psi_1|^2, |\psi_2|^2, |\psi_3|^2$ and $|\psi_4|^2$, the total probability density $|\rho|^2 = |\psi_1|^2 + |\psi_2|^2 + |\psi_3|^2 + |\psi_4|^2$ or the probability current $\vec{j} = (j_x, j_y)^T$ as derived in (2.24), (2.25) and (2.26). As OpenGL has a rotated coordinate system, one of these quantities will be stored in the $y$-component of the vertex vectors:



In order to improve the quality of the rendered scene, we will scale the vertices by a constant factor `scaling`. To visualize the probability density of $\psi_1$, for example, we have to define the three-dimensional vertex vector for each grid point $(i, j)$ as:

$$\texttt{vertex[i,j]} = \begin{bmatrix} x_i \\ \texttt{scaling} \cdot |\psi_1(x_i, y_j)|^2 \\ y_j \end{bmatrix} \tag{4.3}$$

### 4.2.2 Rendering

In order to render a surface we have to combine the previously calculated vertices to triangles. The main reason for doing so is that triangles can be rendered much faster by the GPU than other primitives such as quads or polygons. This however poses some difficulties, our vertices are still in the order of the lattice. We have to come up with an `index` array which defines the precise order in which the vertices are going to be drawn such that 3 subsequent vertices span a triangle. In total we have to draw $6 \cdot (L-1)^2$ triangles. Figure 4.3 shows an example of an `index` array for a $3 \times 3$ grid.

OpenGL 2.0 offers a very efficient way to render large amount of vertex data using vertex buffer objects (VBOs). VBOs can store vertex data (index, position and normal vectors) persistent in the graphics device memory. Therefore, the `index` array will only be calculated once an reused afterwards. The same holds for the $x$ and $z$ components of the vertex vectors (4.3).

### 4.2.3 Lighting and shading

In order get some depth perception, perceive the scene in three dimensions, we have to apply shading. The shading process alters the color of each vertex based on the absolute position and angle to a light source. To calculate the angle each vertex has to be associated with a normal vector. There are many ways of calculating the per vertex normal, we shall take the simplest one as we deal with extremely smooth surfaces. The normal of vertex `(i,j)` is calculated by using the additional grid points `(ia,j)` and `(i,ja)` as defined in equation (4.1) and (4.2). By taking the cross product of this newly defined triangle, as shown below, one can obtain the `normal[i,j]`.

$$\begin{tikzcd} {\texttt{[i,j]}} \arrow[r] \arrow[d] & {\texttt{[i,ja]}} \\ {\texttt{[ia,j]}} & \end{tikzcd}$$

$$\texttt{normal[i,j]} = \vec{a} \times \vec{b}$$

Using the fact that we are dealing with a regular grid with mesh size $\Delta x$, the calculation of $\vec{a}$ and $\vec{b}$ simplifies to:

$$\vec{a} = \begin{bmatrix} \Delta x \\ \texttt{vertex}_y\texttt{[ia,j]} - \texttt{vertex}_y\texttt{[i,j]} \\ 0 \end{bmatrix} \tag{4.4}$$

$$\vec{b} = \begin{bmatrix} 0 \\ \texttt{vertex}_y\texttt{[i,ja]} - \texttt{vertex}_y\texttt{[i,j]} \\ -\Delta x \end{bmatrix} \tag{4.5}$$

After calculating the normals, OpenGL's built-in shaders will take care of calculating the correct darkness of each vertex. To summarize the rendering process:

1. Calculate the vertex vector `vertex[i,j]` for each grid point
2. Calculate the normal vector `normal[i,j]` for each grid point
3. Copy the vertex data to the graphics device and render the scene

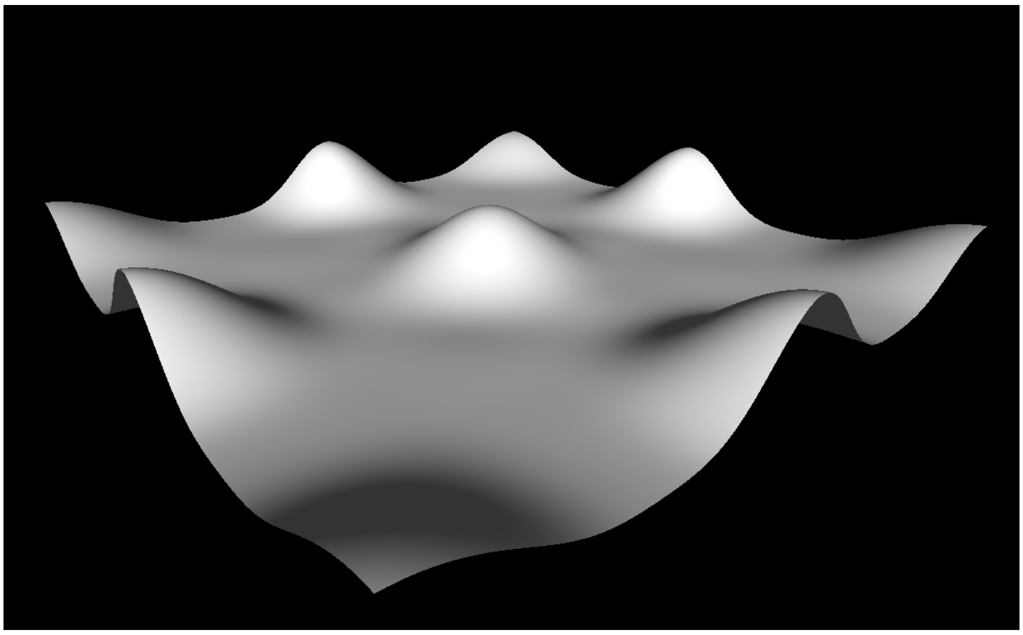


**Figure 4.4:** Visualization of $|\psi_1|^2$ of a free-particle $V \equiv 0$ on a $128 \times 128$ grid rendered as a smooth surface.

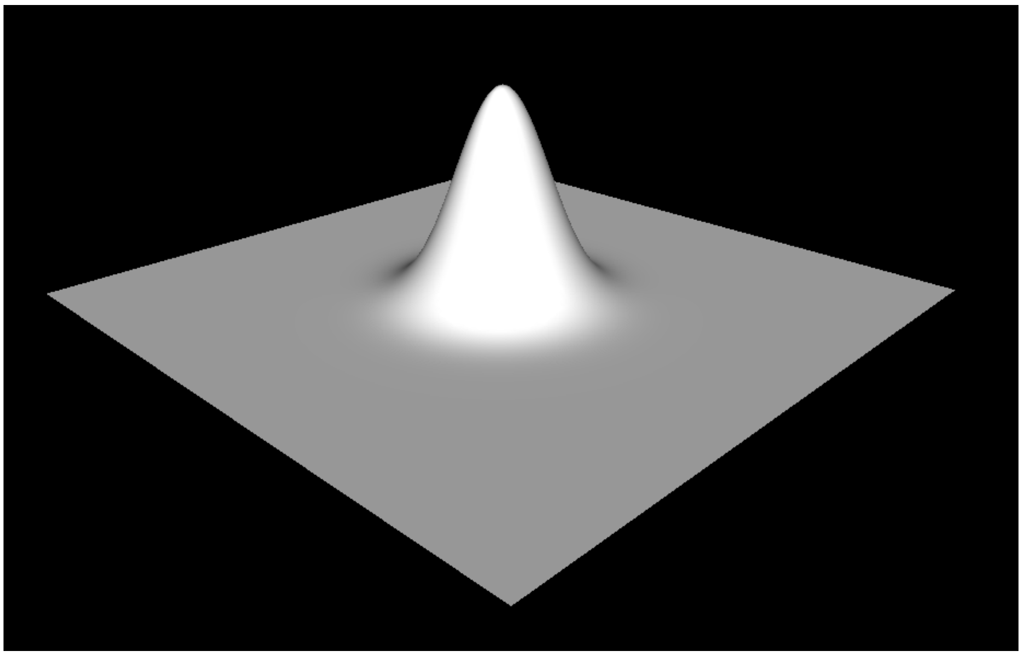


**Figure 4.5:** Visualization of $|\psi_1|^2$ in a harmonic potential (equation 6.2) on a $128 \times 128$ grid rendered as a smooth surface

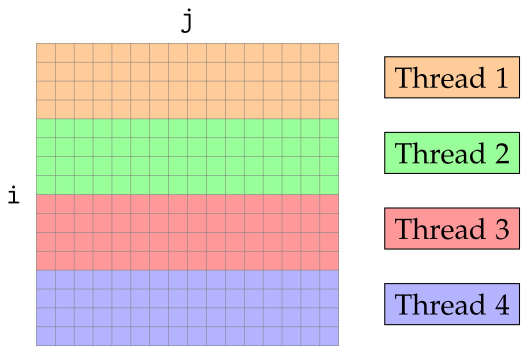


**Figure 4.6:** Domain decomposition of a $16 \times 16$ grid among 4 threads

## 4.3 Parallel implementation

In this section we are going to outline the advantages and caveats which are encountered when porting the QLB scheme to a shared memory multiprocessor. Thus, we are using threads to achieve parallelism. Typically we want to use as many threads as there are physical cores in the system. The QLB program supports threading based on `C++11` threads which are lightweight wrappers around the native thread library of the operating system, like `pthreads` on Linux.

The main idea behind each parallel implementation, is to evenly distribute the work between all threads. Hence, it seems natural for our purpose to simply decompose the grid among all available threads.

### 4.3.1 Domain decomposition

There are many ways of decomposing the grid. The simplest way of doing so, is to divide the rows uniformly among the threads. To divide a $N \times N$ grid among $t$ threads, each thread works on sub-grid of size $\frac{N}{t} \times N$. Therefore, each thread has to do work between the row indices `i_lower` and `i_upper`:

$$\texttt{i\_lower} = \texttt{tid} \cdot \texttt{N/t}$$

$$\texttt{i\_upper} = \begin{cases} \texttt{(tid+1)} \cdot \texttt{N/t}, & \text{if } \texttt{tid} \neq t-1 \\ \texttt{(tid+1)} \cdot \texttt{N/t} \;\; + \;\; \texttt{N\%t}, & \text{else} \end{cases}$$

with $\texttt{tid} = 0, ..., t-1$ representing the thread index. Note that the last thread does potentially some extra work if $N$ is not divisible by $t$. An example decomposition among for 4 threads for a $16 \times 16$ grid is depicted in Fig. 4.6. Obviously, we could have also decomposed along columns but as we are working in a *row-major* storage order, we get better cache locality this way.

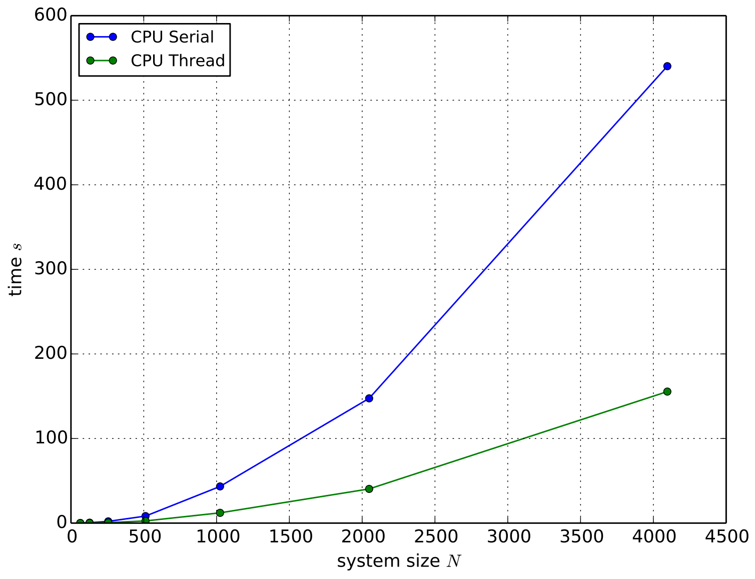


**Figure 4.7:** Runtime of the QLB scheme performing 100 time steps. Measured on an i7-4770k @4.0 GHz running Ubuntu 14.04 and compiled with Clang 3.5.2 (full optimizations). The parallel version was using 4 threads.

### 4.3.2 Synchronization points

After distributing the domain across the different threads, we have to think about possible synchronization among the threads to assert correctness of the algorithm. First, we notice that the rotation and collision steps are embarrassingly parallel, meaning they are complete local to each grid point. The streaming step however requires accessing neighbouring grid points. This might not be a problem, as long as we access grid points in the same row `[i,jk]` and `[i,ja]`. It is most definitely a problem if we access grid points in another row `[ik,j]` and `[ia,j]`, especially when accessing a domain owned by another thread. Hence, we have to insert a barrier before and after the streaming step in $x$-direction (step 3 of the scheme 4.1.3). The barrier will make sure all threads have completed the prior steps. The $y$-direction does not require any synchronization at all as we never access another region than our own.

### 4.3.3 Performance benchmark

**System specification** The performance of the program was measured using a 4-core i7-4770k @4.0 GHz processor running Linux (3.13) with Ubuntu 14.04. The program was compiled using LLVM Clang 3.5.2 with `-O2` as an

| System Size $N$ | Runtime serial | Runtime parallel | Speedup $s$ |
|---|---|---|---|
| 128 | 0.50 s | 0.14 s | 3.57 |
| 256 | 2.05 s | 0.59 s | 3.47 |
| 512 | 8.29 s | 2.56 s | 3.23 |
| 1024 | 43.37 s | 12.13 s | 3.58 |
| 2048 | 147.50 s | 40.43 s | 3.64 |
| 4096 | 540.27 s | 155.50 s | 3.48 |

**Table 4.1:** Runtime and speedup of the QLB scheme performing 100 time steps. Measured on an i7-4770k @4.0 GHz running Ubuntu 14.04 and compiled with Clang 3.5.2 (full optimizations). The parallel version was using 4 threads.

optimization level.

Figure 4.7 shows the runtime for different grid sizes $N$ performing 100 time steps. Table 4.1 shows the speedup of the parallel implementation, with 4 threads, compared to the serial one. The speedup $s$ is calculated as:

$$s = \frac{T_1}{T_p} \tag{4.6}$$

with $T_1$ being the serial runtime and $T_p$ the parallel one. The average speedup is approximately 3.5 which implies a pretty good scaling behaviour as the unobtainable theoretical maximum would be 4. This yields hope that the algorithm will perform well on many-core architectures as explained in the next section.

## 4.4 CUDA implementation

The Compute Unified Device Architecture *CUDA* is a software development platform which allows to write code which can be executed on the graphics processing unit. This enables the user to get access to the massive parallelism intrinsic to GPUs. A graphics processor is designed to have tremendous floating-point performance as well as high memory bandwidth – precisely what is needed in graphics rendering – while the focus on data caching or control flow is minimal. This reflects the fact that computations in computer graphics can be modelled on a data-parallel level, meaning the same section of the program is executed on many data elements in parallel. This massive parallelism is achieved by mapping large sets of data, usually pixels and vertices, to parallel threads.

These specifications limit the realm of possible algorithms which are suited for this purpose. However, the QLB scheme meets all these requirements, the steps in 4.1.3 can be executed fully in parallel while having high arithmetic intensity with a simple control flow.

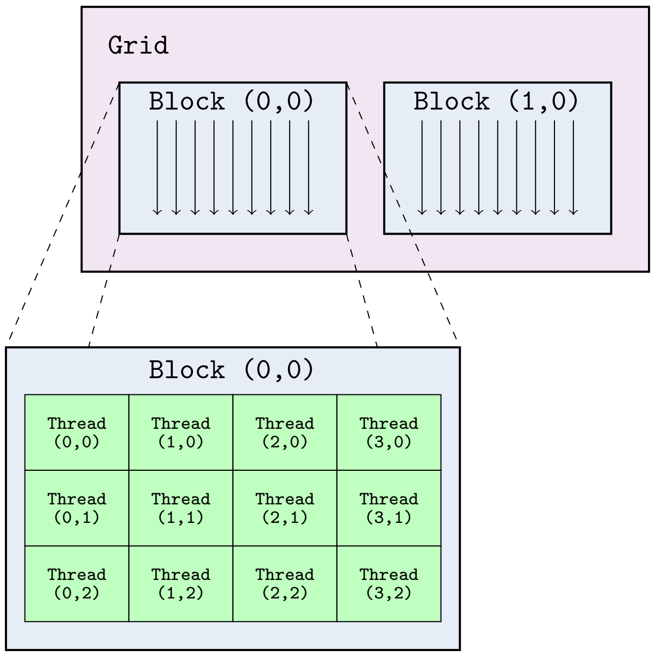


**Figure 4.8:** Example of a two-dimensional grid consisting of 2 two-dimensional blocks with each containing 12 threads.

### 4.4.1 Programming model

The CUDA programming model [1] strives to provide powerful abstractions of the underlying hardware in the form of a C application programming interface. The three key abstractions – a hierarchy of threads, shared memory and barrier synchronization – offer fine-grained data parallelism using thread parallelism.

**Thread hierarchy** The most important concept behind scalable data parallelism is to successively divide a problem, usually represented by a group of operations applied to each element of a large set of data, into smaller pieces. In the CUDA programming model, the first level of abstraction is dividing the problem into smaller sub-problems, referred to as *blocks* of threads. These *blocks* are then organized into a one-, two-, or three-dimensional *grid* of *blocks* (Fig. 4.8). These *blocks* can then be independently scheduled to the different streamed multi-processors (SM) of the device. The sub-problem, assigned to each *block*, is once more segregated into smaller chunks such that it can be solved cooperatively by all threads within the *block*. Usually 32 threads form a team, called a warp, and are executed completely in parallel by the SM, that is, each thread performs the exact same instructions, however on a different portion of the data. The threads are executed by scalar CUDA Cores of the SM. Hence, there is a direct correspondence between the software abstraction and hardware. The number of threads per *block* generally ranges from 32 up 1024 and should always be a multiple of 32, as otherwise some threads are idle. The number of *blocks*, whithin the *grid*, on the other side, is fixed by the problem size.

**Compute model** In general, only a small part of the code is suited to run on the graphics device. This part of the code must be written in CUDA-C/C++ and is usually referred to as a kernel, which is nothing else than a *special* function. The kernel can then be called by the host and will be executed on the device. Both the device and the host have their own separate address space, meaning in order to invoke a kernel on a specific piece of host memory, usually abstracted by `C`-arrays, the desired memory needs to be copied to the device and vice-versa to retrieve the result back. These mem-copy operations, and many more, are provided by the CUDA runtime API.

As pointed out above, a kernel invocation can be seen as a function call with an additional *kernel-launch-configuration*. For example, given a kernel taking two `C`-arrays, `A` and `B`, as parameters, the invocation would look as follows:

```
kernel<<<grid, block>>>(A, B);
```

The `<<<...>>>` syntax specifies the kernel launch configuration which corresponds directly to the hierarchy of threads. The struct `grid`, represented by three integer values, encodes the grid dimensions, meaning the number of *blocks* in each dimension `x,y,z`. The variable `block` specifies the number of threads per *block*, likewise given by three integers. To obtain the configuration showcased in Fig. 4.8 the variables `grid` and `block` would be initialized as:

```
grid.x = 2;         block.x = 4;
grid.y = 1;         block.y = 3;
grid.z = 1;         block.z = 1;
```

The choice of optimal kernel launch configurations is absolutely crucial to obtain good performance as elaborated in 4.4.3.

### 4.4.2 Implementation of the QLB kernels

In this section the implementation process of the QLB scheme in CUDA is going to be outlined. In principle it boils down to provide a kernel for each step in 4.1.3. However, step 1 and 5, as well as 4 and 8, perform the same operations, albeit with different matrices, and hence the same kernels can be used for these steps. The simplest approach to gain data parallelism, is to assign a thread to each grid point $(i, j)$ in the lattice, consequently we just remove the outer loops over the lattice. Doing so is straightforward but will most likely not turn out satisfactory. Indeed, benchmarking both implementations shows that the CUDA version seems to be only twice as fast as the multi-threaded CPU version, although the GPU has roughly 10 times the amount of floating point performance. Hence, there is still a lot of room for optimizations. We will divide the optimization approaches in two parts: optimizing the CUDA code and optimizing the kernel launch configurations. The rest of this section will be spent examining the different techniques which were used to optimize the CUDA kernels in the form of a case study of the rotation kernel.

We start out the optimization process by profiling the four kernels, using the NVIDIA Nsight™ profiler, which delivers the following result:

| Name | Time |
|---|---|
| `kernel_rotate` | 1140 µs |
| `kernel_rotate_back` | 1003 µs |
| `kernel_collide_Q_X` | 539 µs |
| `kernel_collide_Q_Y` | 524 µs |

**Table 4.2:** NVIDIA Nsight profiling results for one invocation of the kernels using a Gefroce GTX 770 and CUDA 7.0 simulating a $512 \times 512$ lattice.

Intuitively one would expect the stream and collision steps would take up the most time, as they have very high arithmetic intensity. However, the rotation kernel, which implements step 1 and 2 in 4.1.3, consumes an unusual amount of time, although it is just a simple $4 \times 4$ matrix multiplication. We will now take a first look at the rotation kernel:

```
void kernel_rotate(cuFloatComplex* spinor,
                   cuFloatComplex* spinorrot,
                   cuFloatComplex* spinoraux,
                   cuFloatComplex* Rinv)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;

  if(i < N && j < N)
  {
    for(int mk = 0; mk < 4; ++mk)
      spinorrot[at(i,j,mk)] = make_cuFloatComplex(0.f, 0.f);

    for(int mk = 0; mk < 4; ++mk)
      for(int nk = 0; nk < 4; ++nk)
        spinorrot[at(i,j,mk)] += Rinv[mk*4 + nk] *
                                 spinor[at(i,j,nk)];

    for(int mk = 0; mk < 4; ++mk)
      spinoraux[at(i,j,mk)] = spinorrot[at(i,j,mk)];
  }
22 }
```

**Listing 4.1:** Original version of the rotation kernel

which implements the rotation step for one single lattice point $(i, j)$. Line 14-17 implement step 1, with `Rinv` being either $X^{-1}$ or $Y^{-1}$ (3.2.1), and line 19 implements step 2 of the scheme. Note that the arrays are accessed through the macro `at(i,j,k)`, with k standing for the 4 components of the spinors, which expands to:

```
#define at(i,j,k) 4*(N*(i) + (j)) + (k)
```

with `N` being the grid length, which just reflects the fact that we use a simple 1D `C-array` for storage but access it as a `N` $\times$ `N` $\times$ 4 tensor. Next, we are going to apply a series of optimizations to this kernel.

### Loop unrolling

While this style of coding was perfectly fine for the CPU version, it is incredibly inefficient on a GPU. The reason being that all the loops have an extreme overhead. In fact, most of the time is spent doing integer arithmetic, hence updating the loop counters. Every modern `C++` compiler would gladly unroll these loops as they have a constant upper bound. However, the CUDA compiler will not automatically do such optimization and therefore we have to do it manually. Further, we can even refine the data parallelism by assigning 4 threads to each lattice point $(i, j)$ such that each thread does the rotation step for one spinor component. This yields:

```
void kernel_rotate_v2(cuFloatComplex* spinor,
                      cuFloatComplex* spinorrot,
                      cuFloatComplex* spinoraux,
                      cuFloatComplex* Rinv)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;
  int k = blockIdx.z*blockDim.z + threadIdx.z;

  if(i < N && j < N)
  {
    spinorrot[at(i,j,k)] =  Rinv[4*k + 0] * spinor[at(i,j,0)];
    spinorrot[at(i,j,k)] += Rinv[4*k + 1] * spinor[at(i,j,1)];
    spinorrot[at(i,j,k)] += Rinv[4*k + 2] * spinor[at(i,j,2)];
    spinorrot[at(i,j,k)] += Rinv[4*k + 3] * spinor[at(i,j,3)];

    spinoraux[at(i,j,k)] = spinorrot[at(i,j,k)];
  }
}
```

**Listing 4.2:** Unrolled loop version of the rotation kernel

Note that we now have to launch three-dimensional blocks of threads, with `block.z = 4`. In addition, the first for-loop (line 11-12 in Listing 4.1) becomes superfluous.

Profiling the kernel again results in:

| Name | Time | Speedup |
|---|---|---|
| Original version (4.1) | 1140 µs | 1.00 x |
| Unrolled loops (4.2) | 600 µs | 1.90 x |

### Optimizing memory access pattern

As already mentioned, the CUDA compiler has limited optimization capabilities compared to normal `C++` compilers. Therefore, another trivial, yet very effective, optimization involves rewriting the lines 12-15 (Listing 4.2) the following:

```
12 spinorrot[at(i,j,k)] = Rinv[4*k + 0] * spinor[at(i,j,0)] +
13                        Rinv[4*k + 1] * spinor[at(i,j,1)] +
14                        Rinv[4*k + 2] * spinor[at(i,j,2)] +
15                        Rinv[4*k + 3] * spinor[at(i,j,3)];
```

Which rather surprisingly improves performance quit significantly:

| Name | Time | Speedup |
|---|---|---|
| Original version (4.1) | 1140 µs | 1.00 x |
| Unrolled loops (4.2) | 600 µs | 1.90 x |
| Improved memory access | 431 µs | 2.65 x |

### Exploiting shared memory

If we recall that each thread is part of a block, and hence shares some per-block local memory with the other threads in the block, we might be tempted to exploit this fact. Taking a closer look at the previous kernel (Listing 4.2) we observe that each thread in the block has to access `Rinv`, more precisely one row. Therefore, it might be beneficial if we *cache* the matrix `Rinv` in a per-block shared memory. This idea can be implemented the following:

```
void kernel_rotate_v3(cuFloatComplex* spinor,
                      cuFloatComplex* spinorrot,
                      cuFloatComplex* spinoraux,
                      cuFloatComplex* Rinv)
```

```
 5 {
 6   __shared__ cuFloatComplex Rinv_loc[16];
 7
 8   int i = blockIdx.x*blockDim.x + threadIdx.x;
 9   int j = blockIdx.y*blockDim.y + threadIdx.y;
10   int k = blockIdx.z*blockDim.z + threadIdx.z;
11
12   int lj = threadIdx.y;
13   int lk = threadIdx.z;
14
15   if(lj < 4 && lk < 4)
16     Rinv_loc[lj*4 + lk] = Rinv[lj*4 + lk];
17
18   __syncthreads();
19
20   if(i < N && j < N)
21   {
22     spinorrot[at(i,j,k)] = Rinv_loc[4*k + 0] * spinor[at(i,j,0)]+
23                            Rinv_loc[4*k + 1] * spinor[at(i,j,1)]+
24                            Rinv_loc[4*k + 2] * spinor[at(i,j,2)]+
25                            Rinv_loc[4*k + 3] * spinor[at(i,j,3)];
26
27     spinoraux[at(i,j,k)] = spinorrot[at(i,j,k)];
28   }
29 }
```

**Listing 4.3:** Shared memory version of the rotation kernel

Note that we have to impose a barrier (line 18) to make sure the local matrix `Rinv_loc` has been successfully initialized. Profiling this kernel version yields:

| Name | Time | Speedup |
|---|---|---|
| Original version (4.1) | 1140 µs | 1.00 x |
| Unrolled loops (4.2) | 600 µs | 1.90 x |
| Improved memory access | 431 µs | 2.65 x |
| Shared memory (4.3) | 362 µs | 3.14 x |

### Precompute indices

After studying the report of the profiler, we still detect a rather high integer arithmetic intensity. This is due to the computation of the indices `at(i,j,k)` which seems to be very costly. Therefore, replacing line 20-28 in Listing 4.3 with precomputed indices:

```
20 if(i < N && j < N)
21 {
22   const int ij  = at(i,j,0);
23   const int ijk = ij + k;
```

```

  spinorrot[ijk] = Rinv_loc[4*k + 0] * spinor[ij + 0] +
                   Rinv_loc[4*k + 1] * spinor[ij + 1] +
                   Rinv_loc[4*k + 2] * spinor[ij + 2] +
                   Rinv_loc[4*k + 3] * spinor[ij + 3];

  spinoraux[ijk] = spinorrot[ijk];
}
```

decreases the run-time even further:

| **Name** | **Time** | **Speedup** |
|---|---|---|
| Original version (4.1) | 1140 µs | 1.00 x |
| Unrolled loops (4.2) | 600 µs | 1.90 x |
| Improved memory access | 431 µs | 2.65 x |
| Shared memory (4.3) | 362 µs | 3.14 x |
| Precomputed indices | 298 µs | 3.84 x |

**Conclusion** If we also apply these optimizations to all the other kernels, we end up with an overall speedup of about 3.5 x (Tab. 4.3). Which impressively shows that writing CUDA code may not be as easy as one might think in the first glance, especially if one is spoilt by the superb optimization mechanisms of normal C++ compilers. The next section will elaborate the other part of the optimization: the choice of the kernel-launch-configurations.

| Name | Old Time | New Time | Speedup |
|---|---|---|---|
| `kernel_rotate` | 1140 µs | 298 µs | 3.83 x |
| `kernel_rotate_back` | 1003 µs | 297 µs | 3.37 x |
| `kernel_collide_Q_X` | 539 µs | 167 µs | 3.22 x |
| `kernel_collide_Q_Y` | 524 µs | 143 µs | 3.66 x |

**Table 4.3:** NVIDIA Nsight profiling results for one invocation of the optimized kernels compared to the original versions (Tab. 4.2) using a Gefroce GTX 770 and CUDA 7.0 simulating a $512 \times 512$ lattice.

### 4.4.3 Choosing optimal kernel-launch-configurations

Choosing the optimal number of threads per block is generally a tricky task, as it is a high dimensional optimization problem. Meaning, it depends on the number of streamed multiprocessors (SM), the available shared memory per SM, the size of the register file and obviously on the kernel itself. The main goal is to assign as many warps (32 threads) to the SM such that we can

| System Size $N$ | Runtime ($16 \times 16$) | Runtime (`QLBoptimizer`) | Speedup |
|---|---|---|---|
| 128 | 0.03 s | 0.01 s | 3.00 |
| 256 | 0.09 s | 0.04 s | 2.25 |
| 512 | 0.39 s | 0.17 s | 2.94 |
| 1024 | 1.55 s | 0.61 s | 2.54 |
| 2048 | 6.28 s | 2.48 s | 2.53 |
| 4096 | 25.46 s | 9.94 s | 2.56 |

**Table 4.4:** Runtime and speedup of the CUDA version of QLB with a static block dimension of $16 \times 16$ threads per block opposed to the block dimensions produced by the `QLBoptimizer`. Benchmarked on a GeForce GTX 770 using CUDA 7.0.

hide the instruction and memory access latency, thus keeping the SMs busy all the time. This behaviour is usually measured in terms of occupancy:

$$Occupancy = \frac{N_{actual}}{N_{max}} \tag{4.7}$$

with $N_{actual}$ being the number of warps per SM and $N_{max}$ the theoretical maximum of warps that fit per SM. The occupancy can easily be retrieved from profiling tools, however this would require to hand-optimize the block dimensions for all system sizes and devices. This is clearly an infeasible task. Therefore, the QLB program comes with an optimizer `QLBoptimizer`, which, once executed, reports the optimal block dimensions for different system sizes and saves them in a configuration file. The optimizer tries out different block sizes, based on some heuristics, and refines the choices in a iterative procedure to eventually converge to an optimal value.

This self-tuning mechanism ensures the best possible run time on all kinds of CUDA devices for the QLB program. Table 4.4 shows the difference between a static block dimension of $16 \times 16$ threads per block, which is according to the CUDA Docs [1] a reasonable starting guess, opposed to the block dimensions selected by the `QLBoptimizer` program.

### 4.4.4 Performance benchmark

In this last section we are going to compare the performance of the three implementations: single threaded CPU (serial), multi-threaded CPU and the optimized CUDA version. We will calculate the speedup $s$ according to:

$$s = \frac{T_s}{T_p} \tag{4.8}$$

with $T_s$ being the run-time of the single threaded CPU version and $T_p$ either the run-time of the multi-threaded CPU or CUDA version. Table 4.5 and Figure 4.9 visualize the comparison:

| System Size $N$ | Runtime serial | Runtime CUDA | Speedup $s$ |
|---|---|---|---|
| 128 | 0.50 s | 0.014 s | 35.71 |
| 256 | 2.05 s | 0.044 s | 46.60 |
| 512 | 8.29 s | 0.17 s | 48.76 |
| 1024 | 43.37 s | 0.61 s | 71.10 |
| 2048 | 147.50 s | 2.48 s | 59.50 |
| 4096 | 540.27 s | 9.94 s | 54.35 |

**Table 4.5:** Runtime and speedup of the different implementation of the QLB scheme

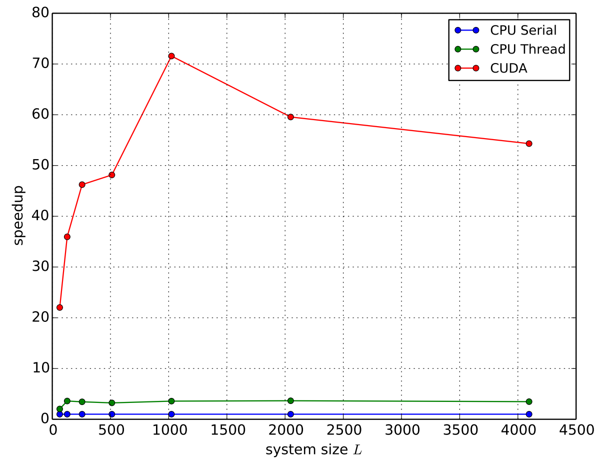


**Figure 4.9:** Speedup (equation 4.8) of the different implementation of the QLB scheme, compared to the single threaded CPU version, performing 100 time steps. Measured on an i7-4770k @4.0 GHz running Ubuntu 14.04 and compiled with Clang 3.5.2 (full optimizations). The parallel CPU version was using 4 threads. The CUDA version was measured on a GeForce GTX 770 using CUDA 7.0.

The CUDA version achieves a speedup of more than one order of magnitude. This convincingly confirms the assumption that lattice Boltzmann schemes scale extremely well on many core architectures [4]. Another reason for those pleasing results is that we never have to do expensive memory copies between CPU and GPU, not even if we visualize the output as CUDA can directly access the OpenGL frame buffers on the device.

This results on the other hand suggest that one could still optimize the CPU versions as a general rule of thumb states that GPU code runs roughly about 5-10 times faster than highly optimized CPU code. Indeed, one could improve the CPU code by manually vectorize it using SIMD instructions, which however would be quit time consuming as there is no support for standard complex arithmetic.

Chapter 5

# Numerical Validation

This chapter will be dedicated to the verification of the implementation. The QLB scheme in general has been verified successfully in the past [8]. We will perform some very basic calculations, namely the free particle motion and a particle in a harmonic potential, and compare them to the results obtained by *Succi et al.* [4]. In both calculations we use a Gaussian wave packet, commonly referred to as *minimum uncertainty* wave packet, as an initial condition for the spin up component of the particle $\psi_1$.

$$\psi_1(x,y,0) = (2\pi\Delta_0^2)^{-1/2} \exp(-\frac{x^2+y^2}{4\Delta_0^2}) \tag{5.1}$$

With $\Delta_0$ representing the initial spread. The other components of the wave function $\psi_2, \psi_3$ and $\psi_4$ are initially set to zero.

## 5.1 Free particle motion

It is generally known, that the motion of a quantum particle in the absence of any potential ($g \equiv 0$ in 2.1) loses coherence. Meaning the spatial extent, the temporal evolution of the standard deviation of $\psi_1$, grows to infinity. Hence, to measure this phenomena, we would like to calculate the spreads:

$$\Delta_x(t) = \left( \frac{\int \psi^\dagger x^2 \psi \, dV}{\int \psi^\dagger \psi \, dV} \right)^{1/2} \tag{5.2}$$

and

$$\Delta_y(t) = \left( \frac{\int \psi^\dagger y^2 \psi \, dV}{\int \psi^\dagger \psi \, dV} \right)^{1/2} \tag{5.3}$$

after each time step. However, the evaluation of an integral on a computer requires the use of a quadrature rule. Which yields, using the trapezoidal

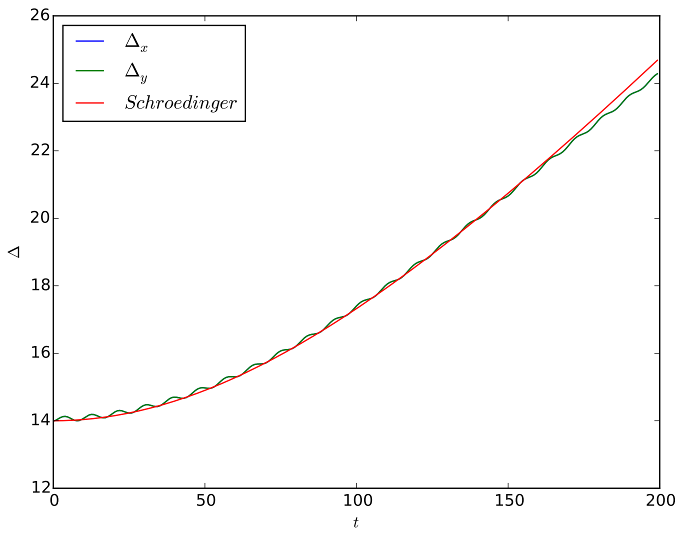


**Figure 5.1:** Evolution of a Gaussian wave packet with initial spread $\Delta_0 = 14$, particle mass $m = 0.35$ discretized on a grid with $L = 128$ grid points in each dimension and mesh width $\Delta x = 0.78125$.

rule:

$$\Delta_x(t) = \left( \frac{\sum_{i,j} \psi^\dagger x_i^2 \psi \, \Delta x^2}{\sum_{i,j} \psi^\dagger \psi \, \Delta x^2} \right)^{1/2} \tag{5.4}$$

and similarly for $\Delta_y$. The analytical solution of the Schrödinger equation for a free particle, with the initial condition given by (5.1), is [4]:

$$\psi_{exact}(x, y, t) = \frac{\exp(-\frac{x^2+y^2}{4\Delta_0^2+2i\hbar t/m})}{(\sqrt{2\pi}[\Delta_0 + \frac{i\hbar t}{2m\Delta_0}])^{3/2}} \tag{5.5}$$

Therefore, the analytical expression for the spreads is given by:

$$\widehat{\Delta_k}(t) = \left( \Delta_0^2 + \frac{\hbar^2 t^2}{4m^2\Delta_0^2} \right)^{1/2} \qquad k = x, y \tag{5.6}$$

**Exeperimental setup** To validate the implementation we measure the spreads (5.2) and (5.3) for a free particle with mass $m = 0.35$ and initial spread $\Delta_0 = 14$ with $L = 128$ grid points in each dimension and associated mesh width of $\Delta x = 0.78125$. Figure 5.1 shows that the simulated spreads are in good agreement with the analytical solution (5.6). This result also matches the outcome of the calculations of *Succi et al.* [4], which was anticipated.

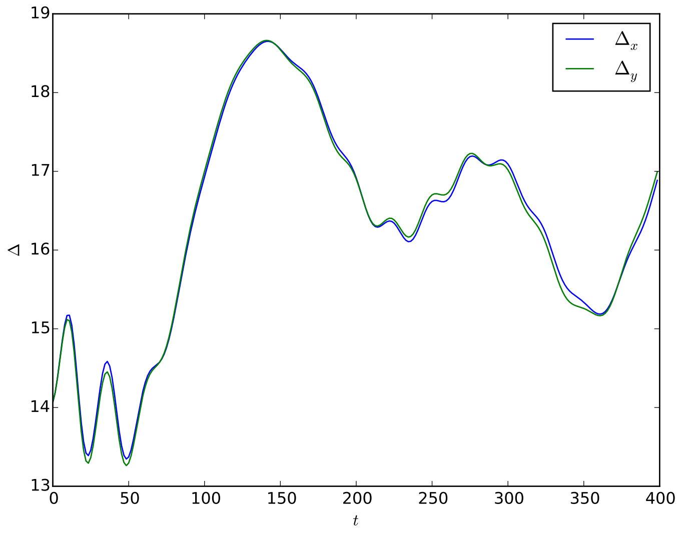


**Figure 5.2:** Evolution of a Gaussian wave packet with initial spread $\Delta_0 = 14$, particle mass $m = 0.1$ discretized on a grid with $L = 128$ grid points in each dimension and mesh width $\Delta x = 1.5625$ in a harmonic potential (5.7).

## 5.2 Harmonic potential

Next, we are going to simulate a particle in a spherically symmetric harmonic potential:

$$V(x,y) = -\tfrac{1}{2}m\omega_0^2(x^2+y^2) \tag{5.7}$$

with

$$\omega_0 = \frac{1}{2m\Delta_0^2} \tag{5.8}$$

being the frequency associated with the harmonic oscillation. With this frequency the initial spread $\Delta_0$ of the wave packet should be conserved. To validate the scheme we set the particle mass $m = 0.1$ and the initial spread to $\Delta_0 = 14$ and use a computational grid given by $L = 128$ grid points in each dimension and a mesh width of $\Delta x = 1.5625$.

Figure 5.2 shows that the spreads $\Delta_x$ and $\Delta_y$ correspond very well to each other but deviate from the initial value $\Delta_0 = 14$. This was also observed in [4] and seems to be due to the sensitive dependence of the solution on the spatial discretization. If we lower the spatial resolution, $\Delta x = 0.78125$, better results are being obtained, as shown in Fig. 5.3.

All of these figures can easily be reproduced using the script `validate.sh` in the QLB source directory.

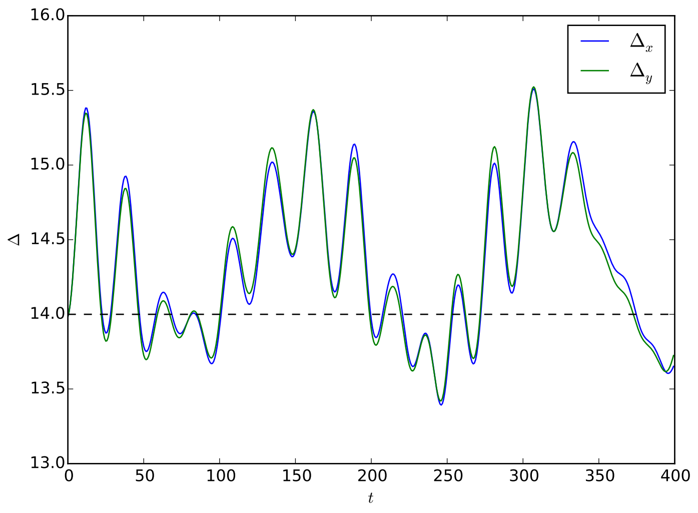


**Figure 5.3:** Evolution of a Gaussian wave packet with initial spread $\Delta_0 = 14$, particle mass $m = 0.1$ discretized on a grid with $L = 128$ grid points in each dimension and mesh width $\Delta x = 0.78125$ in a harmonic potential (5.7).

Chapter 6

# Usage

This chapter will focus mainly on the usage of the program. In the beginning there will be a short explanation of the overall structure of the code, followed by a quick installation guide and an illustration of the available command-line arguments. Last but not least, there will be an example execution of the entire process, hence generating initial condition and potential data and running the program with the appropriate command-line arguments.

## 6.1 Structure of the code

As already pointed out in chapter 4, the code is written in `C++11` with minimal dependencies on external libraries. The project contains about 40 files and roughly 10'000 lines of code. The code is structured in way which separates the user interface and the QLB algorithm. Therefore, the entire code of the QLB scheme is defined in `QLB.hpp` and implemented in `cpp`-files starting with `QLB` e.g `QLBcpu.cpp`, `QLBgraphics.cpp` etc... The header file `QLB.hpp` also contains a documentation of the functions. The user interface files, including window creation and keyboard and mouse input handling, are prefixed with `GLUT`. In addition, there are some miscellaneous files for error handling and utility tasks.

## 6.2 Installation

The project is available on github[1]. There is also a much more comprehensive guide on how to compile the program. The compilation process varies heavily among the different operating systems. Nonetheless, you will always need a compiler tool chain which is capable of compiling `C++11`.

[1]https://github.com/thfabian/QLB

**Linux & Mac OSX** To build QLB you need to make sure the following libraries are installed: libGLEW, libglut as well as the CUDA SDK 7.0 (versions older than CUDA 7.0 might not work as the code makes use of `C++11` which is only partially supported in prior versions). To compile the program from scratch:

1. Obtain the source: `git clone https://github.com/thfabian/QLB`
2. Change into the QLB folder: `cd QLB/`
3. Install libGLEW if you haven't already: `make libGLEW`
4. Compile with `make` which builds QLB by default with CUDA and assumes your CUDA installation is residing in `/usr/local/cuda/`. To disable CUDA run `make CUDA=false` instead.

**Windows** To build QLB on Windows, you should use the Visual Studio 2012 project. There are two build customizations which allow to build QLB with or without CUDA. The CUDA version requires the CUDA SDK 7.0 Visual Studio integration.

There are also precompiled binaries `QLBwin.exe` and `QLBwin-no-cuda.exe` in `bin/Windows/32-bit` and `bin/Windows/64-bit`. The CUDA binaries were compiled with CUDA 7.0 and therefore require a NVIDIA graphics driver of version 346 or higher.

## 6.3 Command-line options

Once QLB is compiled properly, we can control the execution by passing command-line arguments. Many arguments are set to a default value if nothing is specified. The following options are supported:

| **Command-line argument** | **Explanation** | **Default** |
|---|---|---|
| `--help` | Display information about all available options. | - |
| `--version` | Display the version information. | - |
| `--verbose` | Run in verbose mode i.e print useful statistics about the OpenGL and CUDA runtime environment. | - |
| `--no-gui` | Turn off visualization. | - |
| `--fullscreen` | Expand the window to full screen. | - |
| `--potential=S` | See 6.4.1 | `free` |
| `--g=X` | Set the coupling constant $g$ of the built-in potentials. (See 6.4.1) | `1.0` |
| `--initial=S` | See 6.4.2 | - |

| | | |
|---|---|---|
| `--tmax=X` | Perfom `X` timesteps i.e run the simulation in `[0, X·dt]`. | `100` |
| `--L=X` | Set the number of grid points $L$ in each dimension to `X`. | `128` |
| `--dx=X` | Set the spatial discretization $\Delta x$ to `X`. | `1.5625` |
| `--dt=X` | Set the temporal discretization $\Delta t$ to `X`. | `1.5625` |
| `--delta0=X` | Set the initial spread $\Delta_0$ to `X`. (See 6.4.2) | `14` |
| `--mass=X` | Set the mass of the particles $m$ to `X`. | `0.1` |
| `--plot=S` | See 6.3.1 | - |
| `--device=S` | Set the device the simulation will run on. `S` can be one of: `[cpu-serial,cpu-thread,gpu]`. | gpu |
| `--nthreads=X` | Execute the cpu version with `X` threads. | - |
| `--config=S` | Set the configuration file, produced by `QLBoptimizer`, to `S`. | QLBconfig.conf |
| `--dump-at=X` | Dump the state of the simulation at time `X·dt` to a file. (See 6.3.2) | - |
| `--dump-load=X` | Load the dump file `S` to be used with the static viewer . (See 6.3.2) | - |

**Table 6.1:** Command-line options supported by QLB

### 6.3.1 Plotting

QLB offers the ability to write many different quantities to a file after the simulation has finished i.e `tmax` time steps have been performed. They can be specified by passing the option `--plot=S` to QLB with `S` representing a combination of the following:

- `all`: Print all possible quantities.
- `spread`: Print the spreads $\Delta_x$ and $\Delta_y$ after each time step (See 5.1).
- `spinor1`: Print the probability density of the spin up particle $|\psi_1|^2$.
- `spinor2`: Print the probability density of the spin down particle $|\psi_2|^2$.
- `spinor3`: Print the probability density of the spin up anti-particle $|\psi_3|^2$.
- `spinor4`: Print the probability density of the spin down anti-particle $|\psi_4|^2$.

- `density`: Print the total probability density $\rho$ (2.24).
- `currentX`: Print the current in $x$-direction $j_x$ (2.25).
- `currentY`: Print the current in $y$-direction $j_y$ (2.26).
- `veloX`: Print the velocity in $x$-direction $v_x$ (2.28).
- `veloY`: Print the velocity in $y$-direction $v_y$ (2.29).

Multiple options are seprated by `','` e.g `--plot=spinor1,currentX`. Note that this option is only avaible in non-gui mode (requires the argument `--no-gui`).

### 6.3.2 Static viewer

In order to run a large scale simulation offline but still get some visual intel, QLB offers the feature to dump the state of the simulation at any given point in time. The dump file contains the vertex data (vertices and normals) stored in a binary format. The dump file can latter be read by QLB's static viewer which will display it in it's 3D visualization framework. There are two ways of creating a dump file, depending if the simulation is run with or without the gui. Without the gui (`--no-gui`) the command:

```
./QLB --no-gui --tmax=100 --dump-at=100
```

will create a dump file at time `100·dt` i.e after 100 time steps. In gui-mode pressing key `D` will create what we desire. The file will usually be labeled with the grid size $L$ e.g `dump128.bin`. To load the simulation back in the visualization framework, we can use:

```
./QLB --dump-load=dump128.bin
```

## 6.4 Input libraries

The solution of the Dirac equation is entirely dependant on the initial condition and potential which are being used. It seems essential to provide a tool which can easily generate those quantities. For this purpose an input library, with bindings for `python` and `Matlab`, has been written. The library comes with an extensive documentaion e.g in `Matlab` type: `help setPotential` to get detailed help on how to create a potential. In addition, there are some examples which showcase the usage of the library in the folder `input/examples/`.

### 6.4.1 Potential

To specify the potential QLB has the command-line option `--potential=S` with `S` being either one of the four built-in potentials:

- `free`: Free particle motion
$$V(x,y) = 0 \tag{6.1}$$
- `harmonic`: Spherically symmetric harmonic potential
$$V(x,y) = g \cdot (-\tfrac{1}{2}m\omega_0^2(x^2+y^2)) \tag{6.2}$$
with $\omega_0 = \frac{1}{2m\Delta_0^2}$ being the frequency associated with the harmonic oscillation. $\Delta_0$ is the initial spread (6.4.2) and $g$ the coupling constant.
- `barrier`: Rectangle barrier potential
$$V(x,y) = \begin{cases} g \cdot V_0 & \text{if } x < \frac{1}{3}L \cdot dx \text{ and } x > \frac{1}{6}L \cdot dx \\ 0 & \text{else} \end{cases} \tag{6.3}$$
with $V_0 = \frac{L^2}{32\Delta_0^4 m}$ and $g$ the coupling constant.
- `GP`: Gross-Pitaevskii potential
$$V(x,y) = g \cdot (|\psi_1(x,y)|^2 + |\psi_2(x,y)|^2 + |\psi_3(x,y)|^2 + |\psi_4(x,y)|^2) \tag{6.4}$$

or a file with the potential data generated by the input library. The strength of the coupling constant $g$ can be controlled with `--g=X`. An example of how to use the input library to generate the built-in potential `harmonic` is showcased in Listing 6.1.

### 6.4.2 Initial condition

By default, the initial condition of $\psi_1$, the spin-up component of the particle, is a spherically symmetric Gaussian wave packet with initial spread $\Delta_0$:

$$\psi_1(x,y,0) = (2\pi\Delta_0^2)^{-1/2} \exp(-\frac{x^2+y^2}{4\Delta_0^2}) \tag{6.5}$$

The other spin components $\psi_2, \psi_3$ and $\psi_4$ are initially set to 0 and $\Delta_0$ defaults to 14. By passing the option `--initial=S`, with `S` being a file with the initial data, the default condition will be overwritten. Note that the file must be in a specific format (as specified in `inputGenerator.m` or `inputGenerator.py`).

## 6.5 Sample Execution

In this last section, the complete process, creating an initial condition, potential and running the program with the appropriate flags, will be shown. We will use the initial condition (6.5), although for simplicity without the pre factor, and a simple harmonic potential (6.2) with $g = 1$. As a result, we will evolve a wave packet with initial spread $\Delta_0 = 14$, particle mass $m = 0.1$ and the spatial and temporal discretization set to $\Delta t = \Delta x = 1.5625$ discretized on a grid with $L = 128$ points in each dimension.

**Initial condition and Potential** We will use the `Matlab` version of the input library, which is located in the folder `input/InputGenerator.m`. Listing 6.1 demonstrates the usage of the input library which will result, once executed, in the creation of the two input files. To get the full documentation of the functions, type `help setInitial` for example.

```
% Include the path of the library
addpath <path-to-InputGenerator.m>

% Define the system size L, spatial discretization Δx, mass m
% and initial spread Δ0
L      = 128;
dx     = 1.5625;
mass   = 0.1;
delta0 = 14.0;

% Initialize the class
ig = InputGenerator(L, dx, mass, delta0)

% Set the spinor0 component (the others are initialzed with 0 by
% default). Note that spinor0 corresponds to ψ1.
Ifunc = @(x,y) exp(-((x)^2+(y)^2)/(4*delta0*delta0));
ig = setInitial(ig, Ifunc, 0);

% Set the harmonic potential
Vfunc = @(x,y) -1.0/2*mass*(1.0/(2*mass*delta0^2))^2*(x*x + y*y);
ig = setPotential(ig, Vfunc);

% Write the initial condition and potential to the output files
ig = writeInitial(ig, 'myInitialCond.dat');
ig = writePotential(ig, 'myPotential.dat);
```

**Listing 6.1:** Matlab script to create input data using InputGenerator.m

**Running QLB** Now, we want to run the QLB program. Therefore, we have to atleast provide our generated input files:

```
./QLB --potential=myPotential.dat --initial=myInitialCond.dat
```

Further, we could set the temporal discretization $\Delta t = 1.5625$:

```
./QLB --potential=myPotential.dat --initial=myInitialCond.dat
      --dt=1.5625
```

If we want to write the total density $\rho$ (2.24) and the current in $x$-direction $j_x$ (2.25) after 600 time steps to some output files, we have to write:

```
./QLB --potential=myPotential.dat --initial=myInitialCond.dat
      --dt=1.5625 --no-gui --tmax=600 --plot=density,currentX
```

Note that for plotting we need to turn off the visualization `--no-gui`.

Chapter 7

# Conclusions

## 7.1 Conclusions

The work presented in this thesis aims to provide an extensive toolkit to study the two-dimensional Dirac equation. For this purpose a high performance Dirac solver using the QLB scheme has been implemented. The solver was ported to support multi- as well as many core architectures which yielded, especially in the case of the CUDA implementation, significant performance improvements. This approved the general valid belief that lattice Boltzmann schemes are well suited for modern highly parallel computer architectures, even in the quantum case. A simple case study, performed with the rotation kernel, demonstrated the difference between code that is at a proof of concept level opposing to carefully optimized one.

The solver was verified by reproducing results from the literature, to exhibit its correctness. In addition, a real-time visualization framework, based on OpenGL, has been developed to provide an immediate visual feedback of the simulated physics. The code has been written in a manner which will easily allow extending the QLB scheme in the future as the algorithm is condensed in a couple of files.

The resulting program strives to offer an easy-to-use interface to control the execution, as well as simple scripting libraries to create input data.

## 7.2 Future Work

The code has proven to be very efficient and scalable. Hence, a future extension could involve porting the code to distributed systems using MPI. Preliminary steps have already been taken as the mechanisms of dumping the simulation to a binary file would come in very handy in this situation to still get some visual feedback. As most clusters are mainly powered by

CPUs, further improvement attempts could go towards vectorizing the CPU code to speed up the calculation if there is no CUDA device at hand.

Due to the nature of the QLB scheme in higher dimensions, namely the operator splitting approach, an extension to solve the three-dimensional Dirac equation is within the realm of possibilities.

# Bibliography

[1] Nvidia Corporation. *CUDA Toolkit Documentation v7.0*. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, 2015. Accessed: 2015-05-02.

[2] K. S. Novoselov, A. K. Geim, S. V. Morozov, D. Jiang, M. I. Katsnelson, I. V. Grigorieva, S. V. Dubonos, and A. A. Firsov. *Two-dimensional gas of massless Dirac fermions in graphene*. Nature (London) 438, 197, 2005.

[3] L.H. Haddad and L.D. Carr. *The nonlinear Dirac equation in Bose–Einstein condensates: Foundation and symmetries*. Physica D: Nonlinear Phenomena, Volume 238, 2009.

[4] S. Palpacelli, P. J. Dellar, D. Lapitski and S. Succi. *Isotropy of three-dimensional quantum lattice Boltzmann schemes*. Phys. Rev. E83, 046706, 2011.

[5] E. Merzbacher. *Quantum Mechanics* . John Wiley & Sons, New York, third edition, 1998.

[6] F. Fillion-Gourdeau, H. J. Herrmann, M. Mendoza, S. Palpacelli, and S. Succi. *Formal Analogy between the Dirac Equation in Its Majorana Form and the Discrete-Velocity Version of the Boltzmann Kinetic Equation*. Phys. Rev. Lett. 111, 160602, 2013.

[7] S. Palpacelli and S. Succi. *Numerical validation of the quantum lattice Boltzmann scheme in two and three dimensions*. Phys. Rev. E75, 066704, 2007.

[8] S. Succi. *Numerical solution of the Schrödinger equation using discrete kinetic theory*. Phys. Rev. E 53, 1996.

[9] S. Succi. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Oxford University Press, Oxford, 2001.

[10] S. Succi and R. Benzi. *Lattice Boltzmann equation for quantum mechanics*. Physica D: Nonlinear Phenomena, Volume 69, Issues 3–4, 1993.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| | |
| | |
| | |
| | |
| | |

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| | |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*