Solving topological lattice models on coprocessors

Topi Siro

\[ H |\Psi\rangle = E |\Psi\rangle \]

The lowest eigenvalue of \( T \) approximates the ground state energy of \( H \), and since \( T \) is tridiagonal, symmetric and usually quite small, \( \dim T \approx 100 \), it can be easily diagonalized with standard methods. The Krylov space dimension \( m \) does not need to be fixed beforehand. Instead, we can check for convergence against a suitable criterion after each iteration of the loop. For example, one can check the change in the lowest eigenvalue, and deem the algorithm converged, if it falls under some threshold.

We can also construct an approximation for the eigenvector corresponding to the lowest eigenvalue. For this, we need the Lanczos basis vectors \( f_j \). Due to memory constraints, storing them all is often not feasible. Instead, we can run the algorithm twice, starting from the same vector \( f_1 \), so that both runs give the same set of \( f_j \). The first time gives us the tridiagonal matrix \( T \) and the lowest eigenvalue. Then, in the second run we can generate the eigenvector in the original basis, step by step, from the eigenvector of \( T \) and the vectors \( f_j \):

\[
|0_i\rangle = \sum_{j=1}^{m} \lambda_{j} |f_j\rangle
\]

where \( |0_i\rangle \) is the approximated ground state of \( H \) and \( \lambda_{j} \) is the \( j \)th element of the lowest eigenvector of \( T \).
Solving topological lattice models on coprocessors

Topi Siro

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall M1 of the school on 14 October 2016 at 13.

Aalto University
School of Science
Department of Applied Physics
Quantum Many-Body Physics
Supervising professor
Academician, Aalto Distinguished Professor Emeritus Risto Nieminen

Thesis advisor
Adjunct Professor Ari Harju

Preliminary examiners
Prof. Dage Sundholm, University of Helsinki
Dr. Pekka Manninen, CSC-IT Center for Science

Opponent
Dr. Nicolas Regnault, École normale supérieure, France
Abstract

Understanding how the various electronic properties of matter emerge from the motion and interaction of electrons has been an important goal of physics since the early 1900s. One important tool has been the study of quantum lattice models, which can be considered as simplified depictions of solids. Provided that the system is small enough, it is possible to solve the low energy spectrum accurately with numerical methods.

Like most problems in modern physics, studying lattice models requires extensive numerical computation. Traditionally, computer programs have been written to run on the central processing unit, but in recent years, various new parallel computing coprocessors have been introduced. Graphics processing units, which were originally added to render images on the computer screen, can now also be used for general purpose computation. Another new platform is the Xeon Phi coprocessor, specifically designed to accelerate parallel programs. Both of these coprocessors are parallel systems, where there are hundreds or thousands of computational threads running concurrently. This poses challenges in designing and implementing algorithms that benefit from the parallelism.

In this Thesis, we implement the exact diagonalization method on graphics processors and the Xeon Phi. We apply it on topological lattice models, which have been under intense study recently. They feature topological phases that cannot be explained with Landau’s symmetry-breaking theory, but instead require studying the topological properties of the ground state. One key quantity in identifying the phases is the Chern number that is related to the transverse conductance in quantum Hall phases.

In the so called checkerboard model, we show that the topological ground state can withstand strong local impurities. With increasing impurity density, we observe transitions to a metallic state and an insulating state. In another model, the Haldane-Hubbard model, we study the phase diagram with changing on-site interaction and sublattice potential. We find an interesting intermediate topological phase, where the symmetry of the up and down spins breaks spontaneously.

Keywords lattice model, GPU, CUDA, Xeon Phi, Lanczos algorithm

Author
Topi Siro

Name of the doctoral dissertation
Solving topological lattice models on coprocessors

Publisher School of Science

Unit Department of Applied Physics

Series Aalto University publication series DOCTORAL DISSERTATIONS 186/2016

Field of research Theoretical and Computational Physics

Manuscript submitted 18 May 2016 Date of the defence 14 October 2016

Permission to publish granted (date) 13 September 2016 Language English

Monograph Article dissertation Essay dissertation


ISSN-L 1799-4934 ISSN (printed) 1799-4934 ISSN (pdf) 1799-4942

Location of publisher Helsinki Location of printing Helsinki Year 2016

Tiivistelmä

1900-luvun alkupuolelta lähtien yksi fysiikan suurimpia päämääriä on ollut ymmärtää, miten aineen elektroniset ominaisuudet syntyvät sen elektronien liikkeestä ja vuorovaikutuksesta. Eräitä tärkeitä tutkimuskohteita ovat olleet kvanttihilamallit, joita voidaan pitää kiinteän aineen yksinkertaistettuina kuvauksina. Jos systeemit ovat tarpeeksi pieniä, niiden matalan energian spektri voidaan tutkia tarkasti numeerisin menetelmin.


Avainsanat: hilamalli, GPU, CUDA, Xeon Phi, Lanczos-algoritmi

ISSN-L: 1799-4934
ISSN (painettu): 1799-4934
ISSN (pdf): 1799-4942
Julkaisupaikka: Helsinki
Painopaikka: Helsinki
Vuosi: 2016
Sivumäärä: 99
This thesis is the product of my doctoral studies in the Quantum Many-Body Physics group at the Department of Applied Physics, Aalto university School of Science between the years 2011 and 2016.

First of all, I would like to express my deepest gratitude to Adjunct Professor Ari Harju for supervising my doctoral studies. His quirky humour, relaxed yet professional attitude and astonishing Emacs skills create a very pleasurable atmosphere in the group. I would also like to thank Academician, Aalto Distinguished Professor Emeritus Risto Nieminen for supervision and all of his work for the infrastructure of the Aalto physics department.

I have had the pleasure of being surrounded by very talented and friendly people during the years in the group, and I would like to thank all current and former members of QMP. Special thanks belong to Dr. Mikko Ervasti for coercing me into drinking coffee and for making a lot of it. Many thanks to Academy Professor Päivi Törmä, M. Sc. Tuomas Vanhala, Dr. Zheyong Fan and Dr. Long Liang for fruitful collaboration. I also wish to thank Dr. Mari Ijäs, Dr. Tuukka Hiltunen, Dr. Andreas Uppstu, Dr. Marc Dvorak and M. Sc. Ville Vierimaa.

Aside from the research work, I would like to thank Dr. Aki Kutvonen, an old friend and colleague, for all those lunches and discussions on the life of a doctoral candidate. Many thanks to Dr. Ville Havu for all those courses I had the pleasure of assisting in and Dr. Ivan Degtyarenko for giving me the opportunity to teach GPU computing for new students.

For financial support, I thank the Finnish Doctoral Programme in Computational Sciences and Tekniikan edistämissäätiö. I would also like to thank the Aalto Science-IT project and the CSC-IT Center for Science for providing the computing resources.

Special thanks go to my wife Emmi Ruokokoski for being the sunshine
of my life. Last but not least, I thank my family and friends for making my life so awesome.

Helsinki, August 31, 2016,

Topi Siro
## Contents

**Preface**  

**Contents**  

**List of Publications**  

**Author’s Contribution**  

**1. Introduction**  

**2. Exact Diagonalization**  

2.1 Second quantization  

2.2 The Lanczos algorithm  

**3. Computing with Coprocessors**  

3.1 Graphics processors  

3.1.1 Kernels  

3.1.2 Memory  

3.1.3 GPU programming  

3.2 Xeon Phi  

**4. Implementation**  

4.1 CUDA implementation  

4.1.1 Storing the Hamiltonian  

4.1.2 Lanczos implementation  

4.2 CPU and Xeon Phi implementations  

4.3 Benchmarks  

**5. Topological lattice models**  

5.1 Computing the Chern number  

5.2 Checkerboard model
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “Exact diagonalization of the Hubbard model on graphics processing units”

The author performed all numerical simulations, analyzed the data, prepared the figures and wrote the manuscript.

Publication II: “Exact diagonalization of quantum lattice models on coprocessors”

The author performed all numerical simulations, analyzed the data, prepared the figures and wrote the manuscript.

Publication III: “Impurities and Landau level mixing in a fractional quantum Hall state in a flatband lattice model”

The author performed all numerical simulations, analyzed the data, prepared the figures and wrote most of the manuscript.

Publication IV: “Topological phase transitions in the repulsively interacting Haldane-Hubbard model”

The author performed most of the exact diagonalization calculations for the phase diagram, formulated the ansatz and did the overlap calculations and participated in writing the manuscript.
1. Introduction

Quantum mechanics revolutionized physics in the beginning of the 20th century. The deterministic paths of particles in Newtonian mechanics were replaced with complex-valued wave functions, which can be used to compute the probabilities for the outcomes of the measurements of the system. The possible outcomes are determined by the eigenstates of the system, and in principle, the states and the corresponding energies can be solved from the Schrödinger equation, which is an eigenvalue problem for the Hamiltonian operator of the system. [1]

In practice, even solving the ground state analytically from the Schrödinger equation for many interacting particles is only possible in a handful of one-dimensional cases [2]. A notable example is the solution of the one-dimensional Hubbard model [3, 4] by Lieb and Wu [5] using the Bethe ansatz [6, 7].

Generally, the eigenstates and energies are solved numerically. Often, approximations are needed to make the problem easier. Popular numerical methods include the Hartree-Fock approximation [8, 9], dynamical mean field theory [10], density functional theory [11] and quantum Monte Carlo [12]. In this Thesis, we will use the so-called exact diagonalization method (ED) [13], which, as the name implies, is exact up to numerical accuracy. This comes with the cost of having to limit to very small systems, because storing the wave functions requires an exponentially increasing amount of memory. However, even in small systems, we can observe interesting physics and also use the exact results as a benchmark for the other, approximative methods.

Since the invention of the first computers in 1940s, they have been applied to solving problems in physics and mathematics [14]. Since then, the computational power of microprocessors has grown at a steady exponential rate, an observation known as Moore’s law [15]. However, the clock
speeds of the cores in modern central processing units (CPU) have not significantly increased during the last ten years, but they have evolved from single-core to multi-core systems. Further improvements in the computing performance thus require moving from serial to parallel programming.

In parallel programming, there are several computational threads running concurrently. This allows for much faster execution, but only if the problem at hand is suited for parallel computation. It also requires more effort from the programmer to write parallel code, and one has to take care of things like communication between the threads and proper memory management to avoid race conditions between threads. However, the potential reward is a significant speedup compared to the serial code.

A relatively new hardware platform for parallel computing is the graphics processing unit (GPU) [16]. They were introduced in PCs in the 1990s, mostly to render the increasingly demanding video game graphics onto the screen. The modern GPU can also be used for general purpose computation, and GPU vendors have even launched products dedicated for this purpose. Another new coprocessor is the Xeon Phi from Intel [17]. Current hardware features up to 72 cores, and it can be programmed with standard methods such as OpenMP and MPI, which makes porting existing multi-core CPU codes onto the Xeon Phi very easy.

In this Thesis, we utilize NVIDIA’s CUDA programming model that allows the programmer to write low-level code that runs on a GPU. We also use OpenMP to run parallel code on a Xeon Phi. We implement the so-called Lanczos [18] algorithm to compute the low-energy spectrum of various lattice models. They are simplified models of solids, where the emergence of the electronic properties of materials from the motion and interaction of electrons can be studied from first principles.

We will focus on topological lattice models, which feature phases that fall outside the traditional theory of symmetry breaking by Landau [19]. Topological lattice models [20] have been under extensive study in recent years, and their potential applications include quantum computers, where their topological nature could protect the quantum properties from disturbances [21]. Intriguingly, at least some of them can be experimentally realized in cold atomic gases in optical lattices [22–24], offering a highly tunable testbed for observing quantum many-body physics.
2. Exact Diagonalization

A straightforward method to numerically solve the Schrödinger equation is to construct the Hamiltonian matrix in some basis and diagonalize it. This is the essence of the exact diagonalization (ED) method, also known as configuration interaction (CI) in quantum chemistry [9]. Pioneering works in the field include the computations on quantum spin models in one dimension by Bonner and Fisher [25] in 1964 and in two dimensions by Oitmaa and Betts [26] in 1978, among others [27–29]. Early results for the Hubbard and t-J models were calculated by Dagotto et al. [30, 31]. In quantum chemistry, the first direct configuration interaction calculations were done by Roos [32] in 1972. For a review of the early developments of computational quantum chemistry, see Reference 33.

The major drawback of the exact diagonalization method is the rapid growth of the basis size, which limits its use to small systems and particle numbers. Several methods have been developed to truncate the basis to conserve memory while trying to preserve the ability to accurately describe the low-energy behaviour of the system [34–37]. In this thesis, we work with the full basis (full configuration interaction), which severely limits the system size but is also straightforward to implement and gives us results with minimal approximations.

2.1 Second quantization

According to the spin-statistics theorem [38, 39], the wave function of many indistinguishable fermions, such as electrons, is antisymmetric under the exchange of two of the particles. This also implies the Pauli exclusion principle [40], i.e. the fact that two fermions cannot simultaneously be in the same single particle quantum state. A convenient way to write
wave functions of many-fermion systems is the Slater determinant \[41\],

\[
\Psi(x_1, x_2, \ldots, x_N) = \frac{1}{\sqrt{N!}} \left| \begin{array}{cccc}
\psi_1(x_1) & \psi_2(x_1) & \cdots & \psi_N(x_1) \\
\psi_1(x_2) & \psi_2(x_2) & \cdots & \psi_N(x_2) \\
\vdots & \vdots & \ddots & \vdots \\
\psi_1(x_N) & \psi_2(x_N) & \cdots & \psi_N(x_N)
\end{array} \right| ,
\]

(2.1)

where \(\psi_i(x_j)\) are the occupied single-particle eigenstates and \(N\) is the number of particles. The antisymmetry of the wave function is guaranteed by the mathematical properties of the determinant.

To further simplify the picture, we use second quantization, which is introduced in detail in Reference 42. To summarize, instead of Slater determinants, we work with Fock states, where we only count the number of particles in each single-particle state. The many-particle state with \(N^\uparrow\) spin-up electrons in single-particle states \(i_0 \ldots i_{N^\uparrow}\) and \(N^\downarrow\) spin-down electrons in states \(j_0 \ldots j_{N^\downarrow}\) can be written as

\[
c_{i_0 \uparrow} \cdots c_{i_{N^\uparrow} \uparrow} c_{j_0 \downarrow} \cdots c_{j_{N^\downarrow} \downarrow} |\Omega\rangle ,
\]

(2.2)

where \(c_{i\sigma}\) are creation operators that create an electron in state \(i\) with spin \(\sigma\). The empty state is denoted by \(|\Omega\rangle\). The state indices in the creation operators are defined to be in ascending order. This is an arbitrary selection but it is important because of the anti-commutation relations of the creation operators and their Hermitian conjugates, the annihilation operators \(c_{i\sigma}\) that annihilate an electron at state \(i\) with spin \(\sigma\) [42]:

\[
\{ c_{i\sigma}^\dagger, c_{j\tau} \} = \delta_{ij} \delta_{\sigma\tau},
\]

(2.3)

\[
\{ c_{i\sigma}^\dagger, c_{j\tau}^\dagger \} = \{ c_{i\sigma}, c_{j\tau} \} = 0.
\]

(2.4)

In this Thesis, we study quantum lattice models, where electrons can be thought to jump between sites of a discrete and static lattice and interact with each other. First, we choose our basis such that the wave function of a single electron in the lattice can be described by a vector \(\psi = (\psi_1, \psi_2, \ldots, \psi_{N_s})\), where \(N_s\) is the number of sites in the lattice and \(\psi_i\) is the probability amplitude for finding the electron at site \(i\). For many electrons, the different elements correspond to the different ways of distributing them into the lattice. For example, the state in Equation (2.2) is one basis state, with the state indices now corresponding to the lattice sites.

We will now move on to constructing the Hamiltonian. Detailed derivations of the kinetic and interacting parts of the Hamiltonian in second
Exact Diagonalization

quantization can be found e.g. in Reference 42. A commonly used model in electronic structure calculations is the tight-binding model, which has been successfully applied to e.g. graphene [43, 44] and related materials [45, 46]. The tight-binding Hamiltonian describes electrons hopping from one lattice site to another, and it can be written as

$$ H_{TB} = \sum_{i,j} \sum_{\sigma=\uparrow,\downarrow} t_{ij} (c_{i\sigma}^\dagger c_{j\sigma} + h.c.), \quad (2.5) $$

where $t_{ij}$ is the hopping amplitude between sites $i$ and $j$ and $h.c.$ stands for Hermitian conjugate.

Adding interactions to the tight-binding Hamiltonian is straightforward. For example, one can add an on-site repulsion between the electrons, introduced by Hubbard and others [3,47–49], and a nearest-neighbour (NN) interaction with

$$ H_{INT} = U \sum_i n_{i\uparrow} n_{i\downarrow} + V \sum_{\langle i,j \rangle} \sum_{\sigma,\tau=\uparrow,\downarrow} n_{i\sigma} n_{j\tau}, \quad (2.6) $$

where $n_{i\sigma} = c_{i\sigma}^\dagger c_{i\sigma}$ is the number operator that counts the number of particles at site $i$ with spin $\sigma$, and $U$ and $V$ are the on-site and NN interaction strengths, respectively. The angled brackets in the second sum mean that the sum is taken over all pairs of neighboring sites.

An important point is that the Hamiltonian commutes with the number operator. This means that the numbers of spin-up and spin-down electrons are conserved. Because of this, we can diagonalize the Hamiltonian separately in all the subspaces with different numbers of particles, greatly reducing the size of the Hilbert space.

For numerical computations, it is convenient to have the Hamiltonian in matrix form. That means computing the matrix elements

$$ H_{ij} = \langle i | (H_{TB} + H_{INT}) | j \rangle. \quad (2.7) $$

Generally, the matrix elements of operators between two Slater determinants can be systematically computed with the so-called Slater-Condon rules [9, 50]. For the interaction part $H_{INT}$, the result is quite simple because in our basis $H_{INT}$ is diagonal. For example, with the Hubbard interaction, the matrix element is the number of doubly occupied sites multiplied by $U$ for each basis state. In the NN interaction case, one counts the number of instances of particles occupying neighboring sites and so on.

The hopping part, $H_{TB}$, on the other hand, is non-diagonal. Computing the matrix elements is a straightforward application of the creation
and annihilation operators, utilizing the anti-commutation relations in Equations (2.3) and (2.4). Intuitively, the result is that there are nonzero elements only between states that can be reached from each other by a single hop of an electron. The absolute value of the matrix element is \( t_{ij} \) and the sign is given by the number of electrons of the same spin that are hopped over. Even and odd numbers give a positive and a negative sign, respectively. This is because we chose to define the basis states as having the creation operators in ascending order in the site index. Concrete examples of forming and storing the basis and the Hamiltonian in the computer memory are given in Chapter 4.

A key property of the Hamiltonian matrix is that it is very sparse, i.e. the fraction of nonzero matrix elements is very small. This is because the number of nonzeros per row is given by the number of available single particle hops, which grows linearly with the lattice size and particle number, while the matrix dimension grows much faster: For lattices with only one orbital per lattice site, the Pauli exclusion principle forbids more than one electron of a particular spin from occupying a site. This means that the Hilbert space dimension is equal to the number of ways of distributing the electrons into the lattice, i.e.

\[
\dim H = \binom{N_s}{N_\uparrow} \binom{N_s}{N_\downarrow}.
\]  

(2.8)

There is also a lot of redundancy in the full hopping Hamiltonian because the up- and down-spins are independent of each other under its action. Because of this, we can consider the full Hilbert space to be a tensor product of two smaller Hilbert spaces for the up- and down-spin particles: \( \mathcal{H} = \mathcal{H}_\uparrow \otimes \mathcal{H}_\downarrow \) and write the hopping Hamiltonian as [51]

\[
H_{TB} = H_\uparrow \otimes I_\downarrow + I_\uparrow \otimes H_\downarrow,
\]  

(2.9)

where \( H_\sigma \) and \( I_\sigma \) are the hopping Hamiltonian and the identity operator for electrons with spin \( \sigma \), respectively. We denote by \( \otimes \) the Kronecker product of matrices, which corresponds to the tensor product of linear maps [52] and is defined as follows: If \( A \) is an \( m \)-by-\( n \) matrix with elements \( a_{ij} \), then the Kronecker product of \( A \) and another (arbitrary sized) matrix \( B \) is a block matrix defined by

\[
A \otimes B = \begin{pmatrix}
    a_{11}B & \ldots & a_{1n}B \\
    \vdots & \ddots & \vdots \\
    a_{m1}B & \ldots & a_{mn}B
\end{pmatrix}.
\]  

(2.10)
Equivalent decompositions have been used in quantum chemistry with so called spin strings [9,53], which offer a compact and convenient way to keep track of which Slater determinants are included in the CI expansion. Decomposing the Hamiltonian like this is extremely useful from the computational point of view, because if the numbers of up- and down-spin particles are similar, $H_{\uparrow}$ and $H_{\downarrow}$ are much smaller than the full Hamiltonian, drastically reducing the amount of memory required to store them.

2.2 The Lanczos algorithm

Our main goal in studying quantum lattice models is to find their eigenenergies and eigenstates, i.e. to diagonalize the Hamiltonian. Analytically, it can only be done for very simple Hamiltonians, such as the one-dimensional Hubbard model [5] and typically various approximations are needed. The main reason is that the Hilbert space size grows exponentially fast as a function of the system size, and with it, the memory requirement to store the wave functions. The first computation to reach a basis size of one billion was done by Olsen et al. [54] and on modern supercomputers basis sizes of over 100 billion have been reported [55]. In practice, half-filled systems, i.e. ones with $N_{\uparrow} = N_{\downarrow} = N_s/2$ are restricted to lattices with around 20 sites if one wants to store the full wave function.

Fortunately, we are mostly interested in the lowest eigenvalues and vectors, which can be computed even for very large sparse matrices. Perhaps the simplest method of obtaining the lowest eigenstate of a Hermitian matrix $A$ is the power method [56, 57], where a random starting vector $x_0$ is repeatedly multiplied by the matrix. Since the vector can be written as a linear combination of the eigenvectors of the matrix, $v_i$,

$$x_0 = \sum_i c_i v_i, \quad (2.11)$$

after $k$ multiplications we have

$$x_k = \sum_i \lambda_i^k c_i v_i, \quad (2.12)$$

where $\lambda_i$ are the eigenvalues of the matrix. After multiple iterations, the term with the highest absolute value $\lambda_i$ starts to dominate. Normalizing after each step, the iteration eventually converges on the dominant eigenvector, which can either have the lowest or the highest eigenvalue. To ensure convergence to the lowest eigenvalue, one can perform the iteration a second time with the matrix $B = A - \lambda_{\max} I$, where $\lambda_{\max}$ is the dominant
Algorithm 1 The Lanczos algorithm [60].

Require: a random initial vector $f_1$ of norm 1

1: $b_1 \leftarrow 0$
2: $f_0 \leftarrow 0$
3: for $j = 1$ to $m$ do
4: $q_j \leftarrow Hf_j - b_jf_{j-1}$
5: $a_j \leftarrow q_j^\dagger f_j$
6: $q_j \leftarrow q_j - a_jf_j$
7: $b_{j+1} \leftarrow \sqrt{q_j^\dagger q_j}$.
8: if $b_{j+1} = 0$ then
9: Stop
10: end if
11: $f_{j+1} \leftarrow q_j / b_{j+1}$
12: end for

eigenvalue of $A$ and $I$ is the identity matrix. This shifts the spectrum such that the lowest eigenvalue is guaranteed to be the dominant one.

While very simple, the power method converges quite slowly [57]. One more sophisticated method for large Hermitian matrices is the Lanczos algorithm [18]. It is a simplified version of the Arnoldi algorithm [58], where eigenvectors of the original large sparse matrix are approximated with vectors in a small subspace. As shown by the power method above, good approximations for the dominant eigenvector can be found in the subspaces of type

$$K_m(f_1, H) = \text{span}(f_1, Hf_1, H^2f_1, \ldots, H^{m-1}f_1),$$

which is called the $m$th Krylov subspace generated by $H$ from a starting vector $f_1$ [59].

The idea of the Lanczos algorithm is to project the Hamiltonian onto an orthogonal basis in a Krylov subspace, which results in a tridiagonal matrix, i.e. one with nonzero elements only on the main diagonal and the first sub- and superdiagonals. The smallest and largest eigenvalues of this tridiagonal matrix turn out to be very good approximations of the corresponding eigenvalues of the Hamiltonian even for quite small values of $m$.

There are multiple versions of the Lanczos iteration, which have been studied by Paige in Reference 61. The most numerically stable one is given in Algorithm 1. It generates the Lanczos basis $\{f_1, f_2, \ldots, f_m\}$ in the Krylov subspace. We form the tridiagonal matrix from the coefficients
Exact Diagonalization

$a_j$ and $b_j$:

$$T = \begin{pmatrix}
    a_1 & b_2 & 0 & \cdots & 0 \\
    b_2 & a_2 & b_3 & \ddots & \vdots \\
    0 & \ddots & \ddots & \ddots & 0 \\
    \vdots & \ddots & b_{m-1} & a_{m-1} & b_m \\
    0 & \cdots & 0 & b_m & a_m
\end{pmatrix}. \quad (2.14)$$

The lowest eigenvalue of $T$ approximates the ground state energy of $H$, and since $T$ is tridiagonal, symmetric and usually quite small, $\text{dim } T \approx 100$, it can be easily diagonalized with standard methods.

The Krylov space dimension $m$ does not need to be fixed beforehand. Instead, we can check for convergence against a suitable criterion after each iteration of the loop. For example, one can check the change in the lowest eigenvalue, and deem the algorithm converged, if it falls under some threshold.

We can also construct an approximation for the eigenvector corresponding to the lowest eigenvalue. For this, we need the Lanczos basis vectors $f_j$. Due to memory constraints, storing them all is often not feasible. Instead, we can run the algorithm twice, starting from the same vector $f_1$, so that both runs give the same set of $f_j$. The first time gives us the tridiagonal matrix $T$ and the lowest eigenvalue. Then, in the second run we can generate the eigenvector in the original basis, step by step, from the eigenvector of $T$ and the vectors $f_j$ [31]:

$$|\Psi_0\rangle = \sum_{j=1}^{m} \tau[j] f_j, \quad (2.15)$$

where $|\Psi_0\rangle$ is the approximated ground state of $H$ and $\tau[j]$ is the $j$th element of the lowest eigenvector of $T$.

In the basic form presented in Algorithm 1, the Lanczos algorithm is only reliable in accurately computing the lowest (and highest) eigenvalue of the Hamiltonian. This is because the algorithm is numerically unstable, as demonstrated in Figure 2.1. Rounding errors cause a loss of orthogonality in the Lanczos basis vectors which leads to eigenvalues suddenly collapsing to lower eigenvalues, and eventually the lowest one. The loss of orthogonality can be avoided by explicitly orthogonalizing the vectors $f_j$ [60, 62], but this is not often feasible because of memory limitations when the Krylov space is large.

One might try to address this problem by restricting the size of the Krylov space and restarting the algorithm upon reaching the maximum size. For example, in the modified Lanczos method [63], only one Lanczos
Figure 2.1. The approximate eigenvalues generated by the Lanczos algorithm. The instability of the algorithm can be seen in the sudden collapses of the eigenvalues after converging.

step is taken, after which the approximative ground state is solved from the resulting $2 \times 2$ matrix and used as the starting vector for the next iteration. Another alternative is the Jacobi-Davidson algorithm [13, 64], which converges faster and avoids the loss of orthogonality. However, the algorithm requires storing all of the basis vectors, which makes it unfeasible for large systems. For a review of other methods that aim to at least partially preserve the orthogonality, see Reference 65.

We choose to use the standard Lanczos iteration without orthogonalization as presented in Algorithm 1, mostly because it is simple and memory efficient to implement while still capable of accurately computing a few of the lowest eigenstates with the following scheme: First, while computing the lowest eigenvalue $\lambda_0$, we stop the iteration once the change in it falls under $10^{-10}$. Then, we run the algorithm again and construct the approximate groundstate $|\Psi_0\rangle$. We check for convergence by requiring that

$$
\| H |\Psi_0\rangle - \lambda_0 |\Psi_0\rangle \| < 10^{-5}.
$$

(2.16)

If the above condition is not fulfilled, we restart the algorithm with $|\Psi_0\rangle$.

In our experience, even for the largest matrices, the ground state typically converges on the first or second time with a Krylov space dimension of under 200.

To accurately compute eigenvalues and vectors above the lowest one, we can run the algorithm multiple times. After a new vector has converged,
we add the term
\[
H_m^{\text{shift}} = \sum_{i=0}^{m-1} \lambda_i |\Psi_i\rangle \langle \Psi_i|
\]
(2.17)
to the Hamiltonian. This will increase the eigenvalues of the previously computed states by \(\lambda_i\), so as long as \(\lambda_i\) is a large enough positive number, the next Lanczos iteration will converge to the \(m\)th eigenstate. With the convergence criterion in Equation 2.16 for each state, we have managed to compute tens of excited states without problem. To make sure that no spurious eigenvectors have been introduced due to the loss of orthogonality, one can always check the orthogonality of the computed approximate eigenvectors.
3. Computing with Coprocessors

3.1 Graphics processors

In computers, calculations have traditionally been done by the central processing unit (CPU). In recent years, however, a contender has emerged from a perhaps surprising direction. Namely, due to their increasingly rapid development, driven by the consumer demand for video games, modern graphics processing units (GPU) are capable of general purpose parallel computation [16, 66].

In contrast with the CPU that executes one or a few instructions at a time, the GPU can run tens of thousands of processes in parallel. This massively parallel execution can lead to significant performance benefits, but only if the problem can be effectively divided into small independent pieces. Fortunately, this is the case for many problems in the computational sciences and also in physics and quantum chemistry GPUs have been successfully used to speed up the solution of countless problems, see e.g. References 67–74.

The development of graphics cards in recent years has been extremely fast, very much more so than the development of CPUs, at least if measured by theoretical peak performance (see Figure 3.1). This has enabled the use of GPUs in general purpose calculations, and indeed there are cards specifically designed for this purpose, such as the Tesla product line from NVIDIA.

The GPU hardware comprises multiple streaming multiprocessors (SM) that each have hundreds of cores with floating point and integer arithmetic units. This leads to a very high floating point performance. For example, the peak theoretical performance for a Tesla K40 GPU is 5 and 1.66 TFLOPS ($10^{12}$ floating point operations per second) in single and dou-
Figure 3.1. Comparison of the theoretical peak performance of GPUs and CPUs.

...ble precision arithmetic, respectively. It has 12 GB of memory and the maximum memory bandwidth is 288 GB/s. One of the biggest challenges in programming GPUs is to be able to efficiently utilize this computational power.

3.1.1 Kernels

The most popular framework for general purpose computation on GPUs has been CUDA by NVIDIA. CUDA is an extension of the C programming language. It allows the programmer to write special functions called kernels that are executed on the GPU. They are characterized by the keyword __global__, in the function declaration. Also, when calling them, one has to specify the launch configuration, i.e. the number of threads and how they are organized, for the kernel with the <<<...>>> syntax.

When called, the kernel is executed in parallel by up to thousands of threads. They are organized in blocks, which form a grid (see Figure 3.2). The threads are guided to operate on different data by the intrinsic variables threadIdx, blockIdx and blockDim, which contain the identification numbers for the thread within the block, the block within the grid and the block size, respectively. Both of them can be up to three dimensional, which can be useful when operating on 2D or 3D data.

Threads within the same block are guaranteed to run on the same multiprocessor and they can be synchronized and share data with each other. However, different blocks are completely independent, and even their or-
**Figure 3.2.** The hierarchy of CUDA threads. They are organized in blocks, which form a grid. [75]

```c
__global__ void foo(int* indata, int* outdata)
{
    // Kernel code
    ...
}

int main()
{
    // Host code
    ...

    // Kernel call
    foo<<<numBlocks, numThreadsPerBlock>>>(indata, outdata);
}
```

**Figure 3.3.** An example of declaring a kernel and calling it from the host program.

The order of execution is undefined. That means that GPUs are best suited for data-parallel problems, where the same operation is repeated for a large number of individual data. Also, the operation should preferably be arithmetically intensive, meaning that the ratio of arithmetic operations and memory operations is large. However, most real-life cases are limited by the memory bandwidth because of low arithmetic intensity.

### 3.1.2 Memory

The GPU is a distinct piece of hardware with its own memory, separated from the host system by the PCI-e bus. This means that all appropriate data needs to be transferred from the CPU to the GPU before running any kernels. This can create a major bottleneck, since the speed of the PCI-
The bus is an order of magnitude lower than the memory bandwidth within the GPU. It is therefore imperative to minimize the data transfer between the CPU and the GPU.

Within the GPU, there are many different types of memory. The most important ones are:

- **Global memory**: This is the main memory of the GPU. It is the one that the host system communicates with via the PCI-e bus. From the GPU cores’ point of view, global memory accesses are quite slow (on the order of hundreds of clock cycles) compared to registers and shared memory accesses (a few clock cycles). In current CUDA architectures, global memory is cached in an L1 cache for each multiprocessor and a unified L2 cache shared by all multiprocessors. The programmer has to explicitly allocate global memory in the host program with `cudaMalloc` and transfer the necessary data onto the GPU with `cudaMemcpy`.

- **Shared memory**: Threads belonging to the same block can use shared memory to communicate with each other. Typically, it is used as a user-managed cache, by first reading data from the global memory into shared memory, performing the computation, and then writing the result back to global memory. Physically, shared memory is part of the same on-chip memory that hosts the L1 cache.

- **Registers**: Variables declared in a kernel are by default stored in per-multiprocessor registers, which are very fast. However, allocating too much registers might result in register spilling, where the excess memory has to be stored in global memory instead, reducing performance drastically.

### 3.1.3 GPU programming

In general, parallel programming is much more difficult than writing serial code. Specifically in the case of GPUs, it is also important to understand some details about the hardware to write well-performing kernels. For a comprehensive guide to CUDA programming, see References 16, 66, 75, 76. Some of the most important things to take into account are:

- **Coalesced memory access**: The most important thing for most ker-
nals is to access the global memory of the GPU in an efficient way. Optimally, contiguous threads access contiguous words in the memory. This is because in current CUDA architectures the memory can only be transferred in chunks on 128 bytes. If the memory accesses are scattered, each thread will invoke a 128-byte transfer, most of which is wasted if the thread only wants one 4-byte float, for example. This leads to drastically reduced effective memory bandwidth.

- **Occupancy:** To ensure that the GPU’s resources are being fully utilized, it is important to manage the allocated shared memory and kernels in a way that allows multiple blocks to run on each multiprocessor. For example, in the Tesla K40 GPU there is 48 kB of shared memory and 64 thousand registers per multiprocessor. There can be up to 16 blocks running on a multiprocessor simultaneously, but if each block uses a significant amount of shared memory and registers, the number might be much lower, meaning that a significant fraction of the cores are idle. The ratio of the number of concurrent blocks and the maximum number is known as the occupancy. It is sometimes beneficial to use less shared memory and accept the slow memory access to the global memory if that means achieving higher occupancy. It is usually not important to achieve 100% but anything below 50% might be problematic.

- **Branching:** On the hardware, the threads are bunched in groups of 32, called warps, that execute one instruction at a time simultaneously. Flow control instructions, such as if statements, may lead to different execution paths for threads in the same warp, which causes serialization. This has implications in designing the kernel code and data storage. For example, the block dimensions should be divisible by 32, and sometimes it is useful to pad data by adding empty space in order to align it with the warp size.

### 3.2 Xeon Phi

A recent addition to the coprocessor scene is the Xeon Phi by Intel [17]. It is based on their many integrated core (MIC) architecture, and comprises 61 CPU cores in the first generation. Like the GPU, it is connected to the host system by the PCI-e bus. At least in principle, it can be thought of...
as a big multicore CPU that one can program with standard paradigms, such as OpenMP and MPI. The main advantage is that existing multicore CPU codes are very easy to port to run on the Xeon Phi. In terms of performance compared to GPUs, there have been mixed results [77–80].

The Xeon Phi model used in our work, the 7120X, has theoretical peak performances of 2.4 and 1.2 TFLOPS in single and double precision, respectively. It has 16 GB of memory with a 352 GB/s bandwidth. Each of its 61 cores support up to four simultaneous threads and 512-bit wide vectors, meaning that they can process sixteen single precision or eight double precision floating point numbers simultaneously.

The second generation of Xeon Phi accelerators, called Knights Landing, was released in June 2016 [17, 81]. It features major improvements, including up to 72 cores with a 2D mesh interconnect instead of the ring interconnect of the first generation. The announced models feature 16 GB of memory with up to 7.2 GB/s bandwidth and 3.45 TFLOPS double precision performance. At the time of writing, there are no published independent benchmarks available yet.
4. Implementation

4.1 CUDA implementation

4.1.1 Storing the Hamiltonian

In Chapter 2, we described exact diagonalization and the Lanczos algorithm in mathematical terms. In this Chapter, we discuss the practical implementation on a GPU. Previously, the Lanczos algorithm has been implemented on GPUs for small dense matrices in a graph bisection algorithm in Reference 82 and for language processing in Reference 83. The CUDA implementation of the matrix-vector product for sparse matrices has been discussed in Reference 84.

In our case, we cannot store the full Hamiltonian matrix, but instead split it into two parts according to Equation 2.9. For constructing and storing the Hamiltonian in the memory, it is convenient to represent the basis states with two binary numbers, one for each spin species, where a set and unset bits correspond to occupied and unoccupied lattice sites, respectively [13, 85].

For example, consider a system with $N_s = 4$ sites with $N_{\uparrow} = 2$ and $N_{\downarrow} = 3$ particles. One basis state is

$$c_{0\uparrow}^\dagger c_{2\uparrow}^\dagger c_{0\downarrow}^\dagger c_{2\downarrow}^\dagger c_{3\downarrow}^\dagger |O\rangle.$$ (4.1)

With the binary number representation, we can write this state as

$$c_{0\uparrow}^\dagger c_{2\uparrow}^\dagger c_{0\downarrow}^\dagger c_{2\downarrow}^\dagger c_{3\downarrow}^\dagger |O\rangle \rightarrow (0101) \times (1101).$$ (4.2)

Note that we use the convention where the site indices run from right to left in the binary representation.
To construct the basis, we need a way to label and organize the different basis states. A straightforward way is to order them according to the size of the binary numbers. In this scheme, we can convert the binary representation into an index $J$ by

$$J = i_\uparrow \left( \frac{N_s}{N_\downarrow} \right) + i_\downarrow,$$

where $i_\uparrow$ and $i_\downarrow$ are the positions of the up and down configuration in an ordered list, starting from 0, of all $N_s$-bit numbers with $N_\uparrow$ and $N_\downarrow$ bits set, respectively. To clarify, in the case above there are 6 possible configurations for the up spins: 0011, 0101, 0110, 1001, 1010, 1100, so 0101 corresponds to $i_\uparrow = 1$. Similarly, 1101 is the third smallest 4-bit number with 3 set bits, so $i_\downarrow = 2$. This gives the basis state index as

$$J = 1 \times \left( \frac{4}{3} \right) + 2 = 6.$$

The labeling scheme can be thought of as looping over all up- and down-spin configurations such that the down-spin index runs fastest, see Table 4.1.

With the basis states labeled, we can construct the many-body Hamiltonian from the single-particle Hamiltonian, which essentially is a description of the lattice by specifying the allowed hops between sites. To construct the many-body hopping Hamiltonians $H_\uparrow$ and $H_\downarrow$, one can simply

<table>
<thead>
<tr>
<th>$\uparrow$</th>
<th>$\downarrow$</th>
<th>$i_\uparrow$</th>
<th>$i_\downarrow$</th>
<th>$J$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1011</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>1101</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>1110</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0101</td>
<td>1101</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>1110</td>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>0110</td>
<td>0111</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
<td>5</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>1100</td>
<td>1110</td>
<td>5</td>
<td>3</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 4.1. A scheme for labelling the basis states for $N_s = 4$, $N_\uparrow = 2$, $N_\downarrow = 3$. States are ordered first according to the up-spin configuration (first column) and then according to the down-spin configuration (second column), in ascending order.
loop over all basis states, and for each one check which single particle hops are allowed. For basis state $J$, this gives all the nonzero matrix elements in row $J$ of the Hamiltonian. The magnitudes of the matrix elements are the hopping amplitudes associated with the hops. The sign is determined by how many particles where hopped over, as explained in Chapter 2.

The resulting hopping matrices are very sparse, so to store them we can use a sparse matrix format instead of the full matrix. Our choice is the ELL format [84] (see Figure 4.1), where the matrix is represented by two smaller matrices, one containing the nonzero elements of each row and the other their column indices. The number of columns in the ELL matrices is given by the maximum number of nonzeros per row. Rows that have fewer nonzeros are padded with zeros. The ELL format is appropriate here because the sparsity structure of our matrices fits the format. That is because there are no large variations in the numbers of nonzeros per row, resulting in little padding.

When storing the ELL matrices in the memory, it is important to consider the memory layout. There is a huge difference between row-major and column-major orders because of the resulting access pattern in the matrix-vector operation. On the GPU, we use column-major order so that threads of the same warp access contiguous words in the memory. On
Implementation

the CPU and the Xeon Phi row-major order is faster because it allows for better vectorization of the instructions on each core.

4.1.2 Lanczos implementation

In implementing the Lanczos algorithm, by far the most important part is the matrix-vector operation where the Hamiltonian operates on a vector. For large systems, the \( Hx \) operation takes up the vast majority (>90%) of the computation time. Other operations in the Lanczos algorithm are simple linear algebra operations: scaling, axpy, dot product and computing the norm of a vector. For all of these, there are readily available optimized library implementations in the CUBLAS library for CUDA GPUs and in the MKL library for CPUs and Xeon Phis.

However, the \( Hx \) operation calls for a custom kernel, because, as explained in Chapter 2, we split the hopping Hamiltonian in two parts to save memory. We can write the operation as

\[
H_{TB}x = (H_\uparrow \otimes I_\downarrow + I_\uparrow \otimes H_\downarrow)x, \tag{4.5}
\]

The effect of operating on a state vector with \( H \) can be understood by considering the vector to consist of \( \dim H_\uparrow \) subvectors of length \( \dim H_\downarrow \). The spin-up configuration stays constant within a subvector. The spin-up part of the Hamiltonian can then be thought to operate on a vector that consists of the subvectors:

\[
(H_\uparrow \otimes I_\downarrow)x = H_\uparrow \begin{pmatrix} x^{(0)} \\ x^{(1)} \\ \vdots \\ x^{(\dim H_\uparrow - 1)} \end{pmatrix}. \tag{4.6}
\]

Correspondingly, the spin-down part of the Hamiltonian operates like a normal matrix-vector product for each of the \( \dim H_\downarrow \) subvectors:

\[
(I_\uparrow \otimes H_\downarrow)x = \begin{pmatrix} H_\downarrow x^{(0)} \\ H_\downarrow x^{(1)} \\ \vdots \\ H_\downarrow x^{(\dim H_\uparrow - 1)} \end{pmatrix}. \tag{4.7}
\]

In Algorithm 2, we give the pseudocode for our GPU implementation of the \( Hx \) operation. In the spin-up part, we are essentially copying parts of the input vector to other parts in the output vector, multiplied by a constant. Therefore, the parallelization strategy is to assign \( \dim H_\downarrow \) threads
Algorithm 2 The GPU kernel pseudocode for operating with the Hamiltonian

Require: vector $y$ initialized to 0
Require: $\text{blockID}$ (the thread block index)
Require: $\text{sv}$ (the subvector index)
Require: $\text{id}$ (the thread index within the subvector)
Require: $\text{gid}$ (the global thread id within the whole vector)
Require: $\text{blockID} < \text{dimUp} \times \text{blocksPerSubvector}$

1: sum $\leftarrow 0$

2: 

3: if $\text{threadIdx}.x < \text{numcolsUp}$ then
4: $\text{Ax}_s[\text{threadIdx}.x] \leftarrow \text{AxUp}[\text{threadIdx}.x \times \text{dimUp} + \text{sv}]$
5: $\text{Aj}_s[\text{threadIdx}.x] \leftarrow \text{AjUp}[\text{threadIdx}.x \times \text{dimUp} + \text{sv}]$
6: end if
7: syncthreads
8: if $\text{id} < \text{dimDn}$ then
9: for $i = 0$ to $\text{numcolsUp}$ do
10: sum $\leftarrow$ sum + $\text{Ax}_s[i] \times \text{x}[\text{Aj}_s[i] \times \text{dimDn} + \text{id}]$
11: end for
12: for $i = 0$ to $\text{numcolsDn}$ do
13: $\text{Aij} \leftarrow \text{AxDn}[i \times \text{dimDn} + \text{id}]$
14: $\text{col} \leftarrow \text{AjDn}[i \times \text{dimDn} + \text{id}]$
15: sum $\leftarrow$ sum + $\text{Aij} \times \text{x}[\text{sv} \times \text{dimDn} + \text{col}]$
16: end for
17: $y[\text{gid}] \leftarrow$ sum;
18: end if

per row that compute the dot product of that row with $x$. In the spin-down part, we perform a normal matrix-vector multiplication in each subvector with $H_\downarrow$.

We launch one thread for each element of the vector. We denote by $\text{Ax}$, $\text{Aj}$, $\text{dim}$ and $\text{numcols}$ with suffixes Up and Dn the ELL data matrix, the ELL indices matrix, the dimension of the hopping Hamiltonian and the number of columns in the ELL matrices for up- and down-spin, respectively. First, each thread computes the subvector index, the index within the subvector and the global id within the whole vector, denoted by $\text{sv}$, $\text{id}$ and $\text{gid}$, respectively. Then, we discard any extra thread blocks that we might have launched.
Each thread has a register variable called \textit{sum}, where they accumulate the result of the corresponding element in the output vector, i.e. \( y[gid] \). First, we operate with \( H_\uparrow \) in the for loop starting on line 9. Since all the threads in a block are using the same matrix element, we first load the \( sv \)th row into shared memory on lines 4 and 5. Note that we have added a synchronization barrier on line 7 because not all threads might participate in the shared memory load. Then, in the for loop starting at line 12, we operate with \( H_\downarrow \) on the \( sv \)th subvector. Finally, we write the result in \textit{sum} into the output vector in global memory.

The kernel in Algorithm 2 does not include the interaction part of the Hamiltonian. However, as discussed in Chapter 2, in our basis the interactions are diagonal, so implementing them amounts to each thread computing the interaction matrix element for the basis state corresponding to their global id. For an example with an on-site Hubbard interaction, see Publication I. Doing all three parts, \( H_\uparrow, H_\downarrow \) and the interaction, in the same kernel is essential to avoid multiple slow accesses to the global memory.

4.2 CPU and Xeon Phi implementations

We have also implemented the Lanczos algorithm on a multi-core CPU and a Xeon Phi. The parallel portions on the code use the OpenMP API that allows the programmer to tell the compiler to parallelize the code with \#pragma directives. The major advantage of this approach is that the low-level details of the parallel implementation are left to the compiler, potentially reducing the programming effort significantly compared to the low-level custom kernel approach for the GPU discussed in the previous Section.

The pseudocode for our CPU and Xeon Phi implementation of the \( H_x \) kernel is given in Algorithm 3. Our parallelization strategy is to distribute the different subvectors among the OpenMP threads. For the matrix-vector products in the spin-down part, we partition \( H_\downarrow \) into blocks of size \( \text{blockx} \times \text{blocky} \). Then, each OpenMP thread computes the result for one row of these blocks. This improves the cache usage compared to just computing one row per thread. In most cases, experimentation showed that \( \text{blockx} = 16 \) and \( \text{blocky} = 8 \) gave the best performance.

The main differences compared to the GPU kernel are that the inner loops are written out explicitly and parallelized with the \#pragma omp
Algorithm 3 The CPU and Xeon Phi pseudocode for operating with the Hamiltonian

Require: vector y initialized to 0

Require: gid (the global thread id within the whole vector)

1: #pragma omp parallel for
2:   for sv=0 to dimUp do
3:     for i=0 to numcolsUp do
4:       idx ← sv*numcolsUp + i
5:     end for
6:     for id=0 to dimDn do
8:     end for
9:   end for
10: 
11: #pragma omp parallel for
12:   for sv=0 to dimUp do
13:     for row=0 to dimDn in steps of blocky do
14:       for col=0 to numcolsDn in steps of blockx do
15:         for r=row to row+blocky do
16:           if r<dimDn then
17:             for c=col to col+blockx do
18:               idx ← r*numcolsDn + c
20:             end for
21:           end if
22:         end for
23:       end for
24:     end for
25:   end for
parallel directive and that the Hamiltonian matrix element indices are computed differently because they have been stored in row-major order, in contrast to the column-major order used in the GPU.

4.3 Benchmarks

We have benchmarked our coprocessor implementations of the Lanczos algorithm against both serial and parallel CPU applications. In Publication I, we compare a Tesla M2070 GPU against a serial program running on a Xeon X5650 in two different lattices, a 1D ring and a 2D square lattice. The results are presented in Figure 4.2. The general result is that the GPU vastly outperforms the serial program when the system size is large enough to utilize the GPUs resources. In both lattices, the GPU achieves speedups of around 120 and 100 in single and double precision, respectively. The result should be viewed as a proof of concept of the applicability of GPUs for ED calculations. To properly benchmark the GPU versus the CPU, a multi-core CPU program is required. Such a comparison is made in Publication II.

In Publication II, we present another set of benchmarks with newer
### Single-precision execution times

<table>
<thead>
<tr>
<th>(N_p)</th>
<th>CPU(s)</th>
<th>PHI(s)</th>
<th>GPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.50</td>
<td>1.31</td>
<td>2.20</td>
</tr>
<tr>
<td>6</td>
<td>12.5</td>
<td>9.97</td>
<td>3.84</td>
</tr>
<tr>
<td>8</td>
<td>207</td>
<td>87.0</td>
<td>40.8</td>
</tr>
<tr>
<td>10</td>
<td>2010</td>
<td>1090</td>
<td>425</td>
</tr>
</tbody>
</table>

### Double-precision execution times

<table>
<thead>
<tr>
<th>(N_p)</th>
<th>CPU(s)</th>
<th>PHI(s)</th>
<th>GPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.58</td>
<td>1.60</td>
<td>2.28</td>
</tr>
<tr>
<td>6</td>
<td>18.6</td>
<td>15.3</td>
<td>5.17</td>
</tr>
<tr>
<td>8</td>
<td>342</td>
<td>199</td>
<td>74.8</td>
</tr>
<tr>
<td>10</td>
<td>3420</td>
<td>2790</td>
<td>712</td>
</tr>
</tbody>
</table>

---

**Figure 4.3.** (top) The execution times of the CPU, the Xeon Phi and the GPU in the checkerboard lattice with different particle numbers. There are equal numbers of spin-up and spin-down particles. (middle and bottom) The speedup factors of the Xeon Phi and the GPU compared to the CPU, computed from the execution times in the tables.
Implementation

Hardware: A Tesla K40 GPU, a 12-threaded Xeon E5-2620v2 CPU and a Xeon Phi 7120X coprocessor. One set of benchmarks is presented in Figure 4.3. We measured the execution times and the speedup factors of the coprocessors compared to the multi-core CPU in running the Lanczos algorithm on a checkerboard lattice model with varying particle numbers. The checkerboard lattice model will be introduced in detail in Chapter 5.

As expected, the CPU is the fastest platform for small particle numbers. For large systems, the GPU performs best, achieving a speedup of around 5 over the multi-core CPU program. The Xeon Phi is never the fastest of the trio, but does outperform the CPU for up to a factor of 2.5 in single precision. Our results indicate that currently GPUs are the best choice for accelerating exact diagonalization calculations in terms of raw performance, while the Xeon Phi offers modest improvements with minimal programming effort.
The quantum Hall effect [86, 87], the quantization of the conductance in a two-dimensional electron system under a perpendicular magnetic field, was the first observed example of a topological quantum state. The eigenstates of the system are organized in very flat bands, called Landau levels. The quantized Hall conductivity can be mathematically attributed to a topological quantity, the Chern number, which corresponds to the number of filled Landau levels [88]. Haldane [89] discovered that an analogous state can also be found in a lattice model without Landau levels. The Haldane model has also been experimentally realized in References 22, 23.

Recently, various lattice models that have ground states analogous to the fractional quantum Hall effect (FQHE) have been discovered [90–95]. These models, also called fractional Chern insulators (FCI), typically have a flat band that resembles a Landau level. When the band is filled to an appropriate fraction, such as \( \nu = \frac{1}{3} \), interactions give rise to a ground state manifold (GSM), a collection of almost degenerate states that is separated by a large gap from the excited states. The GSM has a total nonzero Chern number.

### 5.1 Computing the Chern number

The Chern number is a key topological constant that can be used to identify topological phases. It is always an integer with zero and nonzero values corresponding to topologically trivial and nontrivial states, respectively. Mathematically, the Chern number of an energy band of a tight-binding Hamiltonian is defined as the Berry curvature integrated over the first Brillouin zone [96],

\[
C = \frac{1}{2\pi} \int_{k \in BZ} F(k) d^2k, \tag{5.1}
\]
where the Berry curvature $F(k)$ is defined as

$$F(k) = \frac{\partial}{\partial k_x} A_y(k) - \frac{\partial}{\partial k_y} A_x(k),$$  \hspace{1cm} (5.2)

where $A(k)$ is the Berry connection, computed from the eigenstates in the band, $\Psi(k)$, as

$$A(k) = i \langle \Psi(k) | \nabla_k | \Psi(k) \rangle.$$  \hspace{1cm} (5.3)

In many-body systems, the Chern number can be computed by introducing twisted boundary conditions [97],

$$\Psi(x_j + L_j) = e^{i\theta_j} \Psi(x_j),$$  \hspace{1cm} (5.4)

where $j$ indexes the space dimensions and $L_j$ is the length of the system along direction $j$. In practice, we implement this by multiplying the hopping Hamiltonian matrix element with the appropriate phase factor whenever the hop is over a boundary.

With the twisted boundary conditions, the many-body Chern number can be computed as the total Berry phase divided by $2\pi$ accumulated on the torus defined by $\theta_1, \theta_2 \in [0, 2\pi)$. In practice, this can be done by dividing the torus with a grid and computing the groundstate (or whatever state we are interested in) with the boundary conditions corresponding to each point. Then, one can compute the Berry phase in each grid cell and sum them up.

For an illustration of numerically computing the Chern number, see Figure 5.1. We divide the $\theta_1, \theta_2$ space into a discrete grid of points. Typically, even grids as coarse as 4 by 4 are enough if the Berry curvature is smooth.
enough, which typically is the case if the gap between the ground state and the excited states is large enough. Then, to compute the Berry phase around a grid cell, we compute the ground states in each corner, labeled as $|0\rangle$, $|1\rangle$, $|2\rangle$ and $|3\rangle$ in Figure 5.1. Then, we compute the Berry phase around cell $j$ as

$$B_j = - i \ln(\langle 0|1 \rangle \langle 1|2 \rangle \langle 2|3 \rangle \langle 3|0 \rangle).$$  \hspace{1cm} (5.5)

Finally, we can compute the Chern number by summing the Berry phases in all cells [96]:

$$C = \frac{1}{2\pi} \sum_j B_j. \hspace{1cm} (5.6)$$

### 5.2 Checkerboard model

One extensively studied topological lattice model is the so called checkerboard model [92] (see Figure 5.2). It is a two-band model with the Hamiltonian

$$H_0 = \sum_{\langle j,k \rangle} t_{jk} c_{j}^\dagger c_{k} + \sum_{\langle\langle j,k \rangle\rangle} t'_{jk} c_{j}^\dagger c_{k} + \sum_{\langle\langle\langle j,k \rangle\rangle\rangle} t''_{jk} c_{j}^\dagger c_{k} + \text{H.c.}, \hspace{1cm} (5.7)$$

where the sums run over the nearest-neighbor (NN), next-nearest-neighbor and the next-next-nearest-neighbor sites, respectively. The corresponding hopping amplitudes are $t$, $t'$, and $t''$. The absolute value of the complex phase $\phi_{jk}$ is constant but the sign alternates as shown in Figure 5.2, where the arrows indicate the positive direction.
To make the lowest energy band as flat as possible, one should choose the parameters given in Reference [92]: \( \phi_{jk} = \pi/4, t = 1, t' = \pm 1/(2 + \sqrt{2}) \) and \( t'' = 1/(2 + 2\sqrt{2}) \). Then, to obtain the \( \nu = \frac{1}{3} \) FQH state, we need to add a nearest-neighbour repulsive interaction,

\[
H = H_0 + V \sum_{\langle j,k \rangle} n_j n_k, \tag{5.8}
\]

where \( n_j = c_j^{\dagger} c_j \) is the number operator, as defined in Chapter 2.

With strong enough \( V \), a threefold quasi-degenerate ground state manifold (GSM) emerges in the checkerboard model [93], which is characteristic to a \( \nu = \frac{1}{3} \) FQH state, see Figure 5.3. The topological nature of the GSM can be determined by computing the Chern number. In this case, we find a unit total Chern number for the three groundstates, confirming the \( \nu = \frac{1}{3} \) FQH state.

In Publication III, we have studied how impurities in the lattice affect the \( C = 1 \) phase and have observed phase transitions into a metallic phase and an insulating phase by tuning the strength of the impurities. Previously, attractive impurity potentials have been studied in Reference 98. We used two kinds of impurities: impurity potentials and modified nearest-neighbor hopping amplitudes, both with positive and negative signs. These are added, respectively, by setting \( t_{jk} = s \) for the impurity NN hopping amplitudes ( \( t_{jk} = 1 \) for all other NN hops) and adding
Topological lattice models

terms of the form
\[ H_{j}^{\text{pot}} = pn_j \] (5.9)
to the Hamiltonian. Negative and positive values of \( p \) correspond to attractive and repulsive potentials, respectively. The locations of these impurities have been chosen such that they are evenly distributed in the lattice, as presented in Figure 5.2 for the \( 4 \times 3 \) unit cell lattice.

We identify the different phases mostly by the low-energy spectrum and the Chern number, but also by computing the one-body reduced density matrix [99] (1-RDM), defined by
\[ \rho_{ij} = \langle \Psi_0 | c_i^\dagger c_j | \Psi_0 \rangle, \] (5.10)
where \( | \Psi_0 \rangle \) is the ground state. The 1-RDM eigenstates are called natural orbitals (NO). For trivial uncorrelated single Slater determinant states the eigenvalues of the 1-RDM have values 0 and 1, and for correlated states they fall somewhere in-between, indicating that the state cannot be fully described by single particle physics. The sum of the 1-RDM eigenvalues equals the particle number, and in a FQH state the nonzero eigenvalues are equal to the filling fraction. Thus, by computing the 1-RDM spectrum we obtain information about the quantity of correlations present, and furthermore, how the impurities bind particles from the correlated many-body state.

In Figure 5.4, we show the results of adding up to four impurity hoppings into a \( 4 \times 3 \) unit cell checkerboard lattice with 4 spinless electrons and interaction strength \( V = 3t \). The top plots show the energies as a function of the impurity hopping strength \( s \), and \( n = 1, 2, 3, 4 \) indicates the number of impurity hoppings. On the bottom, we show the 1-RDM spectra at the highlighted points in the top plots.

With one impurity hopping, i.e. \( n = 1 \), the GSM stays intact for \( s > 1 \). With \( s < 1 \), the gap closes but reopens for large negative \( s \). The metallic phase in the middle has Chern number equal to two, so it could be interpreted as two ground state manifolds reorganizing as the hopping changes sign. For large negative \( s \), we again see the 3-fold quasi degenerate GSM with \( C = 1 \).

With two and more impurity hoppings, we see the FQH phase become narrower, and with \( n = 3, 4 \) we observe the insulating phase with strong impurities, characterized by a single ground state with \( C = 0 \), separated by a large gap.

On the bottom of Figure 5.4, we show the spectra of the 1-RDM at the
Figure 5.4. (top) Ten lowest eigenenergies of the checkerboard Hamiltonian. There are \( n = 1, 2, 3, 4 \) added impurity hoppings at \( V = 3t \) on a \( 4 \times 3 \) unit cell lattice. The FQH, metallic and insulating phases are indicated by the different background colors, see the bottom panels. (bottom) The spectra of the one-body reduced density matrices of the ground states at the highlighted points on the top figure plots.
Figure 5.5. (top) Ten lowest eigenenergies of the checkerboard Hamiltonian. There are \( n = 1, 2, 3, 4 \) added impurity potentials at \( V = 3t \) on a \( 4 \times 3 \) unit cell lattice. The FQH, metallic and insulating phases are indicated by the different background colors, see the bottom panels. (bottom) The spectra of the one-body reduced density matrices of the ground states at the highlighted points on the top figure plots.
highlighted points in the top plots. The first half of the eigenvalues are very close to zero, and they correspond to the upper energy band in our two-band model. The unperturbed FQH phase is plotted in black. With strong impurities, we can see the appearance of unit eigenvalues in the 1-RDM spectrum, corresponding to particles that are localized on the impurities. Additionally, with \( n = 3, 4 \), we can clearly see the emergence of the insulating phase as the fully occupied natural orbitals with strong impurities. For a more thorough analysis, see Publication 3.

In Figure 5.5, we show the results of adding up to four impurity potentials into the checkerboard lattice. With one potential, there are no transitions with either attractive or repulsive \( p \). Similar to the case with impurity hoppings, with two potentials we observe the collapse of the gap, and with three and four potentials we see the transition to the insulating phase. However, this only happens with attractive potentials. This can be also seen in the corresponding 1-RDM spectra: attractive potentials tend to localize particles, which leads to fully occupied NOs, while the spectra for repulsive potentials resemble the unperturbed system.

The conclusion from the results presented in Publication 3 is that the \( \nu = \frac{1}{3} \) FQH phase in the checkerboard model is very robust against single impurities of both kind, as expected of a topological state that should be stable against local perturbations. Increasing the number of impurities caused a transition to the metallic state. When the number of impurity hoppings was close to the particle number, a transition to the trivial insulating state was observed with both positive and negative hopping amplitudes. On the other hand, only attractive potentials were able to induce the same transition and the FQH phase was overall more robust against repulsive potentials.

### 5.3 The Haldane-Hubbard model

Another extensively studied topological lattice model is the Haldane model, defined on a honeycomb lattice (see Figure 5.6). The original spinless model, introduced in Reference 89, features a topological ground state with \( C = 1 \) that can be broken into an insulator with a potential difference between the two sublattices. In Publication IV, we study the spinful version of the model with added on-site repulsion, called the Haldane-Hubbard model, with the Hamiltonian \( H = H_k + H_t \), where the kinetic
Figure 5.6. The Haldane model is defined on a honeycomb lattice with nearest and next-nearest neighbor hoppings. The arrows indicate the direction of the positive phase in the complex NNN hoppings.

The Hamiltonian part is

\[ H_k = t \sum_{\langle i,j \rangle, \sigma} c_{i\sigma}^\dagger c_{j\sigma} + t' \sum_{\langle\langle i,j \rangle\rangle, \sigma} \exp(i\phi_{ij}) c_{i\sigma}^\dagger c_{j\sigma}. \tag{5.11} \]

Here \( \langle i, j \rangle \) and \( \langle\langle i, j \rangle\rangle \) denote summation over nearest and next-nearest neighbors on a hexagonal lattice, and \( \sigma \) runs over the two spin components. The phase \( \phi_{ij} \) has a constant absolute value and a sign that depends on the direction of the bond, \( \phi_{ij} = \pm \phi \). The on-site part of the Hamiltonian is

\[ H_l = U \sum_i \left( n_{i\uparrow} - \frac{1}{2} \right) \left( n_{i\downarrow} - \frac{1}{2} \right) + \Delta_{AB} \sum_{i, \sigma} \mathrm{sgn}(i)n_{i\sigma}, \tag{5.12} \]

where \( \mathrm{sgn}(i) \) is +1 for sites \( i \) on sublattice \( A \) and −1 for sublattice \( B \). We set \( t = 1 \), \( t' = 0.2 \) and \( \phi = \frac{\pi}{2} \).

The two terms in the on-site part are the Hubbard repulsion and a staggered on-site potential, respectively. Their effects are somewhat opposite: the Hubbard repulsion tries to prevent particles from occupying the same site, while the sublattice potential tries to pull all particles to the sites with negative potential, regardless of their spin. It is this competition combined with the topologically nontrivial groundstate of the kinetic Hamiltonian that produce interesting physics.

Recently, intermediate phases, i.e. ones that reside in the crossover region between two competing phases, have been discovered in various lattice models. In the ionic Hubbard model, there have been multiple different predictions on the nature of an intermediate phase between Mott and band insulator phases, depending on the kinetic Hamiltonian [100–104]. The model has been also experimentally realized [24], but no intermediate phase was observed. In the Haldane-Hubbard model, the existence
of an interesting intermediate phase has been suggested by mean field studies [105–108], but the strong interactions raise concerns about their reliability.

In Publication IV, we show very convincing evidence of the existence of the intermediate phase in the Haldane-Hubbard model by using three different and complementary methods. We study the phase diagram at half-filling with varying $\Delta_{AB}$ and $U$. In addition to exact diagonalization, we also use mean-field and dynamical mean-field calculations to confirm our results. The main result is the phase diagram, presented in Figure 5.7. There are four different regions:

**The quantum Hall phase:** At the origin, the model is in a quantum Hall phase that was found in the original paper by Haldane in the spinless model. Since we have two noninteracting spin channels each in a $C = 1$ phase, the resulting phase has $C = 2$.

**The band insulator:** With $U = 0$, the quantum Hall phase at the origin transitions into a band insulator at $\Delta_{AB} = 1$. This transition was also discovered in Reference [89] in the spinless model. The qualitative picture is that the strong sublattice potential binds the electrons to the sites with the attractive potential, leading to an insulator.

**The Mott insulator:** With strong $U$, we find the antiferromagnetic
Mott insulator, where the strong on-site repulsion between the electrons prevents doubly occupied sites and inhibits the electrons’ motion.

• **The symmetry broken $C = 1$ phase**: The most interesting region in the phase diagram is the intermediate phase under the $\Delta_{AB} = \frac{U}{2}$ line. Here, the competition between the sublattice potential, trying to draw the particles to the attractive sublattice, and the Hubbard interaction, trying to prevent double occupation, causes the symmetry between the spin channels to break. One spin channel is mostly in the band insulator phase ($C = 0$), occupying the attractive sublattice. This effectively cancels the effect of $\Delta_{AB}$, since the other spin channel now feels the Hubbard repulsion on the attractive sublattice. The other spin channel thus prefers the quantum Hall phase with $C = 1$.

To confirm the qualitative picture of the $C = 1$ phase, we have computed the overlaps of the exact diagonalization ground states with the ansatz

$$
|\psi\rangle = \frac{1}{\sqrt{2}} \left( |\text{QH}\rangle_\uparrow |\text{BI}\rangle_\downarrow + |\text{BI}\rangle_\uparrow |\text{QH}\rangle_\downarrow \right),
$$

(5.13)

where $|\text{QH}\rangle$ and $|\text{BI}\rangle$ are the single-component ground states of the noninteracting model for vanishing and large $\Delta_{AB}$, respectively. The overlaps are presented in Figure 5.8. As expected, the overlap is mostly close to zero except in the region under the $\Delta_{AB} = \frac{U}{2}$ line, where it reaches 0.5, confirming our qualitative picture. More analysis on the $C = 1$ phase with other methods is also presented in Publication IV.
Topological lattice models
6. Summary

In this Thesis, we discussed the implementation of the exact diagonalization method on graphics processors and Xeon Phi coprocessors. The core of the method is the Lanczos algorithm that can be used to accurately compute the lowest eigenenergies and eigenstates of the Hamiltonian. The main focus was on the function that operates with the Hamiltonian matrix on a vector, because for large Hamiltonians it is by far the most time consuming part of the Lanczos iteration. The implementation was applied to solving the low-energy spectrum and identifying topological phase transitions in two topological lattice models, the checkerboard model and the Haldane-Hubbard model.

In Publication I, we present our implementation of the Lanczos algorithm on CUDA-enabled GPUs. A low-level custom kernel for operating with the Hamiltonian on a vector is presented. The program was benchmarked against a serial CPU program and excellent performance was observed, showing that GPUs are well-suited for exact diagonalization calculations.

In Publication II, we perform the same benchmark for newer GPU hardware and compare it to a multi-core CPU application and also a Xeon Phi coprocessor. The codes for the multi-core CPU and the Xeon Phi are parallelized with OpenMP. Here, the GPU performs best for large systems, reaching speedup factors of around 5 against the CPU. The Xeon Phi outperforms the GPU only for very small systems, but in those cases the multi-core CPU is the overall fastest platform. However, the Xeon Phi does offer speedups of up to 2.5 compared to the CPU with minimal programming effort.

In Publication III, we study the effect of adding impurity potentials and hoppings into a checkerboard lattice. The model supports the $\nu = \frac{1}{3}$ fractional quantum Hall phase, and we find transitions to a metallic phase.
and an insulating phase as the density and strength of the impurities are increased. The phases are identified by computing the Chern number, the low-energy spectrum and the one-body reduced density matrix. Overall, the topological phase seem to be very robust against isolated impurities.

In Publication IV, we study the phase diagram of the Haldane-Hubbard model as a function of the sublattice potential and the on-site Hubbard interaction. We find four different regions: the quantum Hall phase with Chern number $C = 2$, a trivial band insulator, a Mott insulator and a symmetry broken intermediate phase with $C = 1$. The phase diagram is computed with three different methods, exact diagonalization, mean field and dynamical mean field theory, which give largely agreeing results. Most importantly, the intermediate phase is found by all methods.

The main conclusions from the work presented in this Thesis are that exact diagonalization is an excellent method for observing topological phase transitions in flatband lattice models and that coprocessors, especially GPUs, are well-suited for the calculations, offering significant speedups compared to multi-core CPU codes. Additionally, the Xeon Phi coprocessor was found to offer moderate performance gains compared to the multi-core CPU with very little programming effort.


Bibliography


The lowest eigenvalue of $T$ approximates the ground state energy of $H$, and since $T$ is tridiagonal, symmetric and usually quite small, $\dim T \gg 100$, it can be easily diagonalized with standard methods. The Krylov space dimension $m$ does not need to be fixed beforehand. Instead, we can check for convergence against a suitable criterion after each iteration of the loop. For example, one can check the change in the lowest eigenvalue, and deem the algorithm converged, if it falls under some threshold.

We can also construct an approximation for the eigenvector corresponding to the lowest eigenvalue. For this, we need the Lanczos basis vectors $f_j$. Due to memory constraints, storing them all is often not feasible. Instead, we can run the algorithm twice, starting from the same vector $f_1$, so that both runs give the same set of $f_j$. The first time gives us the tridiagonal matrix $T$ and the lowest eigenvalue. Then, in the second run we can generate the eigenvector in the original basis, step by step, from the eigenvector of $T$ and the vectors $f_j$:

$$|0_i = \sum_{j=1}^{m} \lambda_j |j\rangle f_j,$$

where $|0_i$ is the approximated ground state of $H$ and $\lambda_j$ is the $j$th element of the lowest eigenvector of $T$. In the basic form presented in Algorithm 1, the Lanczos algorithm is only reliable in accurately computing the lowest (and highest) eigenvalue of the Hamiltonian. This is because the algorithm is numerically unstable, as demonstrated in Figure 2.1. Rounding errors cause a loss of orthogonality in the Lanczos basis vectors which leads to eigenvalues suddenly collapsing to lower eigenvalues, and eventually the lowest one. The loss of orthogonality can be avoided by explicitly orthogonalizing the vectors $f_j$ [60, 62], but this is not often feasible because of memory limitations when the Krylov space is large.

One might try to address this problem by restricting the size of the Krylov space and restarting the algorithm upon reaching the maximum size. For example, in the modified Lanczos method [63], only one Lanczos