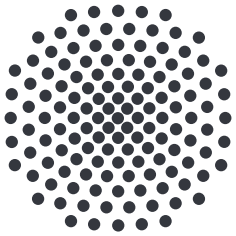
Massively parallel computations of the Bose-Hubbard model with time-dependent potentials

Bachelor thesis of
Kirill Alpin

August 15, 2016

Supervisor: Prof. Dr. Jörg Main



1. Institut für Theoretische Physik
Universität Stuttgart
Pfaffenwaldring 57, 70550 Stuttgart

Contents

1. Introduction

Moore's Law [1] states that the density of transistors on computer chips doubles every 18 months. Every exponential growth in nature cannot continue endlessly (except the expansion of the universe for all we know so far [2]), so cannot Moore's Law [3]. The size of a transistor would eventually reach the size of an atom. What does this mean for computational execution times? The clock speed of a computer chip does not only depend on the switching time of transistors but also on the length of the connections between them. If transistors are small, the connections between them are as well, which decreases their resistance. In turn the computer chip dissipates less energy, so the applied voltage and clock frequency can be increased [4]. A program that contains many computations which depend on previous calculations has to be traversed serially. This means if Moore's Law stalls, so does the runtime of such programs. Otherwise if the program has many separate operations, one can execute these portions in parallel on different points on a computer chip. This explains the recent rise of computational power of graphics cards over CPUs [5], which are inherently designed for parallel tasks. This gain of performance, which is mainly measured in floating point operations per second (flops), can be seen in figure 1.1.

This development has led to an increasing use of these devices not only in computer graphics but also in scientific research as general purpose computing machines. Many computational problems in different kinds of fields of science can be parallelized. These fields include the simulation of quantum mechanical systems, such as the Bose-Hubbard model [7].

The Bose-Hubbard model is the discrete description of a quantum mechanical many-particle system inside an optical lattice. Approximations underlie this model, in which it is assumed that the temperature of the system is at absolute zero and that the particles inside its wells are only allowed to tunnel to neighbouring sites.

The main goal of the present work is to show how to simulate the Bose-Hubbard model with an arbitrary (non-static) potential by exploiting the parallelizability of the underlying algorithm. The resulting speed-up compared to sequential programs is crucial to simulate systems with high numbers of particles and potential wells, as every extra degree of freedom increases the dimension of the Hilbert space exponentially.

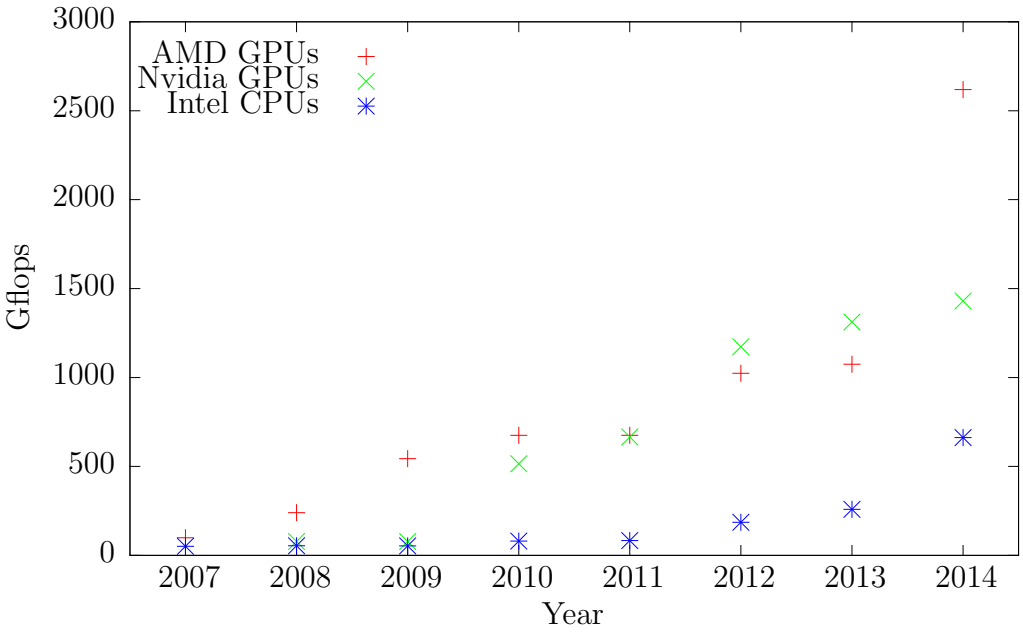


Figure 1.1.: Theoretical double precision performance of GPUs and CPUs over time. It should be noted that it is more difficult to reach the theoretical peak performance of GPUs than CPUs. The reason behind that are primarily hardware restrictions to the way a program is executed on the GPU compared to a CPU. The difference is still significant enough, to outweigh this difficulty. The data shown here is taken from [6].

2. The Bose-Hubbard model

The Bose-Hubbard model [7] is used to study effects of systems with indistinguishable, interacting bosons inside a periodic potential. The physical implementation is usually an optical lattice filled with ultracold bosonic atoms. Following is an outline of the derivation of the Bose-Hubbard model starting with a many-particle Hamiltonian which describes interacting bosons.

2.1. Second quantization

The derivation of the second quantization depicted in this section is based on [8]. The starting point is the general many-particle Schrödinger equation

$$\left[\sum_n \left(-\frac{\hbar^2}{2m}\Delta_{x_n} + V_e(x_n)\right) + \frac{1}{2}\sum_{n,m} V_i(x_n - x_m)\right]\psi = i\hbar\frac{\partial}{\partial t}\psi, \tag{2.1}$$

where $\psi = \psi(x_1, ..., x_N, t)$ depends on the particle coordinates and the time. Such a Hamiltonian contains terms for N non-interacting bosons, namely the kinetic energy of the particles and the external potential $V_e(x_n)$. To take interactions between these particles into account, one introduces a two-body potential $V_i(x_n - x_m)$ which depends on the distance of the two particles.

Attention must be paid to the fact that the Hilbert space described by the Hamiltonian of this system contains physically identical states. As the particles are indistinguishable, the order of the coordinates x_n can be exchanged, which would not alter the physical state of the system. Using this symmetry of the Hamiltonian, it is possible to reduce the Hilbert space to only contain states, which are physically unique.

These states are superpositions of all possible permutations of the order in which the particles appear in the Hamiltonian. One must bear in mind that the phase of the operation P_{nm}, which swaps two particles equals 1 for bosons. These symmetric states can be constructed in the following way,

$$|\psi\rangle = \frac{C}{\sqrt{N!}}\sum_{\text{all } P} P\,|\phi(x_1, ..., x_N)\rangle\,. \tag{2.2}$$

The sum is taken over all possible unique particle permutations of the form

$$P = P_{n_1 m_1} P_{n_2 m_2} ... P_{n_z m_z} \tag{2.3}$$

with 'unique' meaning that permutation operators cannot be transformed by $P_{ab} = P_{ba}$, $[P_{ab}, P_{cd}] = 0$ and $P_{ab}^2 = 1$ into one another. The total number of these permutations is $N!$, which is the reason for the factor $\frac{1}{\sqrt{N!}}$. However, this is still not sufficient to keep the state normalized if particles are located at the same place, i.e. their single-particle wave-functions are equal. Consider the action of P_{12} on a product state $|\phi(x_1)\phi(x_2)\rangle$. The result would be equal to the initial state. The symmetrization of this state is shown in the following formula,

$$\begin{aligned} |\psi\rangle &= \frac{C}{\sqrt{2!}}(I + P_{12}) \, |\phi(x_1)\phi(x_2)\rangle \\ &= \frac{2C}{\sqrt{2}} \, |\phi(x_1)\phi(x_2)\rangle \, . \end{aligned} \tag{2.4}$$

If $|\phi(x_1)\phi(x_2)\rangle$ is and $|\psi\rangle$ has to be normalized, C must have a value of $\frac{1}{\sqrt{2}}$. In general, if n_1 particles are in state $|1\rangle$ and n_2 particles in $|2\rangle$ etc., then

$$C = \frac{1}{\sqrt{\prod_i n_i!}} \tag{2.5}$$

normalizes the state $|\psi\rangle$. Now one can try to artificially construct this Hilbert space for the Hamiltonian in Eq. (2.1), although there is a simpler way to achieve this. By using the local oscillator algebra, the foundation of which are the creation and annihilation operators, the Hilbert space is automatically restricted to symmetrical states. Consider the action of n creation operators on the ground state of a harmonic oscillator,

$$(a^\dagger)^n \, |0\rangle = \sqrt{n!} \, |n\rangle \tag{2.6a}$$

$$\Rightarrow |n\rangle = \frac{1}{\sqrt{n!}} (a^\dagger)^n \, |0\rangle \, . \tag{2.6b}$$

To keep the resulting state $|n\rangle$ normalized, the factor $\frac{1}{\sqrt{n!}}$ must be applied, which is equal to C for n particles at one site. Using this, the creation (annihilation) operator can be described as an addition (subtraction) of one particle at a specific place. To make more clear at which position the particle is added or removed, $a^\dagger$ and a get a subscript x. This means that the position state of a single particle is defined by

$$|x\rangle = a_x^\dagger \, |0\rangle \, . \tag{2.7}$$

For N particles one finds that

$$|x_1, ..., x_n\rangle = \frac{1}{\sqrt{N!}} \left(\prod_{n=1}^{N} a_{x_n}^\dagger \right) |0\rangle \tag{2.8}$$

spans a Hilbert space, which consists of symmetric states in respect to the exchange of particles. Using Eq. (2.8) it is possible to express the Schödinger equation (2.1) in a simplified form. The first few steps of the derivation are shown below. For now the interaction term $V_i(x_n - x_m)$ will be neglected.

$$\begin{aligned}
&\sum_n \underbrace{\left(-\frac{\hbar^2}{2m}\Delta_{x_n} + V_e(x_n)\right)}_{H_p(x_n)} \psi(x_1, ..., x_N, t) \\
=&\sum_n H_p(x_n) \langle x_1, ..., x_N|\psi\rangle \\
=&\sum_n H_p(x_n) \frac{1}{\sqrt{N!}} \langle 0| \prod_m a_{x_m}|\psi\rangle \\
=&\frac{1}{\sqrt{N!}} \langle 0| \sum_n H_p(x_n) \prod_m a_{x_m}|\psi\rangle
\end{aligned} \tag{2.9}$$

The goal now is to develop H_p in the basis of the creation and annihilation operators and then exchange the position of the resulting operator and the product $\prod_m a_{x_m}$. By doing so, the bra becomes $\frac{1}{\sqrt{N!}} \langle 0| \prod_m a_{x_m}| = \langle x_1, .., x_N|$ once again. To achieve this, a trick must be used, where the a_{x_n} is pulled from the product. This is possible, because the commutator $[a_x, a_y] = 0$ holds. Therefore

$$\begin{aligned}
&\frac{1}{\sqrt{N!}} \langle 0| \sum_n H_p(x_n) a_{x_n} \prod_{m\neq n} a_{x_m}|\psi\rangle \\
=&\frac{1}{\sqrt{N!}} \langle 0| \int_{-\infty}^{\infty} \mathrm{d}z \sum_n H_p(z)\delta(x_n - z) a_z \prod_{m\neq n} a_{x_m}|\psi\rangle \,.
\end{aligned} \tag{2.10}$$

Because $\delta(x_n - z) = \left[a_{x_n}, a_z^\dagger\right]$, Eq. (2.10) delivers with some rearranging

$$\frac{1}{\sqrt{N!}} \langle 0| \int_{-\infty}^{\infty} \mathrm{d}z \sum_n \left[a_{x_n}, a_z^\dagger\right] \prod_{m\neq n} a_{x_m} H_p(z)\, a_z|\psi\rangle \,. \tag{2.11}$$

Now to find out what $\sum_n \left[a_{x_n}, a_z^\dagger\right] \prod_{m\neq n} a_{x_m}$ is, consider the first two terms of the sum

$$\left[a_{x_1}, a_z^\dagger\right] a_{x_2}...a_{x_N} + \left[a_{x_2}, a_z^\dagger\right] a_{x_1} a_{x_3}...a_{x_N} + ... \,. \tag{2.12}$$

It can be shown, that $\left[a_x, a_z^\dagger\right]$ commutes with a_y, which leads to

$$\begin{aligned}
&\left[a_{x_1}, a_z^\dagger\right] a_{x_2}...a_{x_N} + a_{x_1} \left[a_{x_2}, a_z^\dagger\right] a_{x_3}...a_{x_N} + ... \\
=&\left(\left[a_{x_1}, a_z^\dagger\right] a_{x_2} + a_{x_1} \left[a_{x_2}, a_z^\dagger\right]\right) a_{x_3}...a_{x_N} + ... \,.
\end{aligned} \tag{2.13}$$

With the operator chaining rule

$$[AB, C] = A[B, C] + [A, C]B \tag{2.14}$$

the expression $\left(\left[a_{x_1}, a_z^\dagger\right] a_{x_2} + a_{x_1}\left[a_{x_2}, a_z^\dagger\right]\right)$ can be simplified to $\left[a_{x_1} a_{x_2}, a_z^\dagger\right]$. If the steps (2.12), (2.13) and the chaining rule are applied recursively on Eq. (2.12), the whole sum reduces to just one commutator,

$$\sum_n \left[a_{x_n}, a_z^\dagger\right] \prod_{m\neq n} a_{x_m} = \left[\prod_m a_{x_m}, a_z^\dagger\right]. \tag{2.15}$$

Using Eq. (2.15), Eq. (2.11) becomes

$$\frac{1}{\sqrt{N!}} \langle 0| \int_{-\infty}^{\infty} \mathrm{d}z \left[\prod_m a_{x_m}, a_z^\dagger\right] H_p(z)\, a_z |\psi\rangle\,. \tag{2.16}$$

The left-side action of $a_z^\dagger$ on $\langle 0|$ is zero, i.e. $\langle 0|\, a_z^\dagger = 0$. So only the positive term of the commutator remains,

$$\begin{aligned}
&\frac{1}{\sqrt{N!}} \langle 0| \int_{-\infty}^{\infty} \mathrm{d}z \prod_m a_{x_m}\, a_z^\dagger H_p(z)\, a_z |\psi\rangle \\
=&\frac{1}{\sqrt{N!}} \langle 0| \prod_m a_{x_m} \int_{-\infty}^{\infty} \mathrm{d}z\, a_z^\dagger H_p(z)\, a_z |\psi\rangle \\
=&\langle x_1, ..., x_N| \int_{-\infty}^{\infty} \mathrm{d}z\, a_z^\dagger H_p(z)\, a_z |\psi\rangle \\
=&\langle x_1, ..., x_N| \int_{-\infty}^{\infty} \mathrm{d}z\, a_z^\dagger \left(-\frac{\hbar^2}{2m}\Delta_z + V_e(z)\right) a_z |\psi\rangle\,.
\end{aligned} \tag{2.17}$$

Now the many-particle Hamiltonian in the basis of the creation and annihilation operators consists of only one integral. This form of the Hamiltonian is named the second-quantized Hamiltonian. The derivation of the second quantized form of the interaction term goes in a similar manner as above. In general a n-body interaction term involves n integrals. The resulting Hamiltonian with particle interaction reads

$$H = \int_{-\infty}^{\infty} \mathrm{d}z a_z^\dagger \left(-\frac{\hbar^2}{2m}\Delta_z + V_e(z)\right) a_z + \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathrm{d}z\, \mathrm{d}w a_z^\dagger a_w^\dagger V_i(z-w) a_w a_z. \tag{2.18}$$

2.2. Fock states

There is still a problem with the current form of the Hamiltonian (2.18). It allows for uncountably infinite basis-states to exist, whereas bound states are known to be a countable set of vectors. So the next step is to reduce the allowed set of vectors from an uncountable to a countable one by projecting the creation and annihilation operators

$a_z^\dagger$ and a_z onto a countable infinite complete orthonormal set of functions $\phi_k(x)$. This means the functions have the following properties,

$$\int_{-\infty}^{\infty} dx \phi_k(x) \phi_{k'}^*(x) = \delta_{kk'}, \tag{2.19a}$$

$$\sum_k \phi_k(x) \phi_k^*(x') = \delta(x - x'). \tag{2.19b}$$

The projection of $a_z^\dagger$ onto these functions is performed as follows,

$$\begin{aligned} a_z^\dagger &= \int_{-\infty}^{\infty} \mathrm{d}z' \delta(z - z') a_{z'}^\dagger \\ &= \int_{-\infty}^{\infty} \mathrm{d}z' \left[\sum_k \phi_k(z) \phi_k^*(z') \right] a_{z'}^\dagger \\ &= \sum_k \phi_k(z) \int_{-\infty}^{\infty} \mathrm{d}z' \phi_k^*(z') a_{z'}^\dagger \\ &= \sum_k \phi_k(z) b_k^\dagger, \end{aligned} \tag{2.20}$$

where the operator

$$b_k^\dagger = \int_{-\infty}^{\infty} \mathrm{d}z' \phi_k^*(z') a_{z'}^\dagger \tag{2.21}$$

obeys the ordinary commutator relations for discrete creation and annihilation operators. The proof for $\left[b_k, b_{k'}^\dagger \right] = \delta_{kk'}$ is shown below,

$$\begin{aligned} \left[b_k, b_{k'}^\dagger \right] &= \int_{-\infty}^{\infty} \mathrm{d}z \, \mathrm{d}w \phi_k(z) \phi_{k'}^*(w) a_z a_w^\dagger - \int_{-\infty}^{\infty} \mathrm{d}z \, \mathrm{d}w \phi_k^*(z) \phi_{k'}(w) a_z^\dagger a_w \\ &= \int_{-\infty}^{\infty} \mathrm{d}z \, \mathrm{d}w \phi_k(z) \phi_{k'}^*(w) (a_z a_w^\dagger - a_w^\dagger a_z) \\ &= \int_{-\infty}^{\infty} \mathrm{d}z \, \mathrm{d}w \phi_k(z) \phi_{k'}^*(w) \delta(z - w) \\ &= \int_{-\infty}^{\infty} \mathrm{d}z \phi_k(z) \phi_{k'}^*(z) \\ &= \delta_{kk'}. \end{aligned} \tag{2.22}$$

With $[a_z, a_w] = [a_z^\dagger, a_w^\dagger] = 0$ one can conclude that the same holds for $b_k^\dagger$ and b_k. These operators can be interpreted as an addition (subtraction) of a particle with a single-particle wavefunction $\phi_k(x)$ to (from) the system. There is a useful representation of

states that exploits this interpretation,

$$|n_1, n_2, ...\rangle = \frac{1}{\sqrt{\prod_k n_k!}} \prod_k (b_k^\dagger)^{n_k} |0\rangle \,, \tag{2.23}$$

which is called Fock state. The numbers n_k describes how many particles are populating the state $\phi_k(x)$.

2.3. Cold atoms in an optical lattice

To proceed further, assumptions have to be made regarding the system one wants to examine. The first assumption is, that the two-particle interaction can be modeled as hard-spheres with diminishing sizes, i.e. a delta potential $V_i(x-y) = F\delta(x-y)$. F can be adjusted in such a way that the scattering cross-section for the real potential of an atom and this potential are the same, which results in an approximation of the real potential for long distances. The second assumption is, that the external potential $V_e(x)$ can be represented as a tunable sine-like potential, which corresponds to the light-dipole interaction with a standing wave, which can be constructed by two oppositely directed lasers. The third assumption is, that the bosons have a temperature of $T = 0\,\mathrm{K}$, so there are no single-particle excitations, i.e. the particles exist all in the lowest Bloch band [9].

Applying Eq. (2.20) to the non-interacting part of the Hamiltonian, Eq. (2.18) becomes

$$\begin{aligned}
&\int_{-\infty}^{\infty} \mathrm{d}z a_z^\dagger \left(-\frac{\hbar^2}{2m}\Delta_z + V_e(z) \right) a_z \\
&= \int_{-\infty}^{\infty} \mathrm{d}z \sum_k \phi_k^*(z) b_k^\dagger \underbrace{\left(-\frac{\hbar^2}{2m}\Delta_z + V_e(z) \right)}_{H_l(z)} \sum_{k'} \phi_{k'}(z) b_{k'} \\
&= \sum_{kk'} b_k^\dagger b_{k'} \underbrace{\int_{-\infty}^{\infty} \mathrm{d}z \phi_k^*(z) H_l(z) \phi_{k'}(z)}_{H_{kk'}} \,.
\end{aligned} \tag{2.24}$$

It is assumed that the optical lattice is deep enough, so that a set of highly localized functions can be used as basis. That means, that the overlap of these functions at non-neighbouring sites are negligibly small, which ensures the sparsity of the Hamiltonian. These functions $w_{nk}(x-ak)$, known as Wannier functions [10], span the whole new Hilbert space by translation, with a being the periodicity of the lattice and n denoting the band index or excitation of a single particle. As we are only interested in the lowest band, these can be dropped, which results in the representation

$$\phi_k(x) = w(x - ak). \tag{2.25}$$

As stated above, the functions $w(z-ak)$ are highly localized, so terms in the sum over k and k' where $|k-k'|>1$ can be dropped, because the hopping integral $J_{kk'}$ is in those cases negligibly small. So $H_{kk'}$ becomes

$$H_{kk'} = V_k\delta_{k,k'} - J_{kk'}(\delta_{k-1,k'} + \delta_{k+1,k'}) \tag{2.26}$$

with

$$V_k = \int_{-\infty}^{\infty} \mathrm{d}z w^*(z-ak)H_l(z)w(z-ak), \tag{2.27a}$$

$$J_{kk'} = -\int_{-\infty}^{\infty} \mathrm{d}z w^*(z-ak')H_l(z)w(z-ak). \tag{2.27b}$$

With $V_i(x-y) = F\delta(x-y)$ the interacting term of the Hamiltonian (2.18) yields

$$\begin{aligned} I &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathrm{d}z\,\mathrm{d}w a_z^\dagger a_w^\dagger F\delta(z-w)a_w a_z \\ &= F\int_{-\infty}^{\infty} \mathrm{d}z a_z^\dagger a_z^\dagger a_z a_z. \end{aligned} \tag{2.28}$$

By applying the same procedure as for the non-interacting part, one arrives at the following expression for the second term in Eq. (2.18),

$$I = F\sum_{klmp} b_k^\dagger b_l^\dagger b_m b_p \underbrace{\int_{-\infty}^{\infty} \mathrm{d}z w^*(z-ak)w^*(z-al)w(z-am)w(z-ap)}_{I_{klmp}}. \tag{2.29}$$

Through the localisation of $w(z)$, an approximation of I_{klmp} is given by considering only terms with $k=l=m=p$, so Eq. (2.29) becomes

$$I = \sum_k \frac{U_k}{2} b_k^\dagger b_k^\dagger b_k b_k \tag{2.30}$$

with

$$U_k = 2F\int_{-\infty}^{\infty} \mathrm{d}z |w(z-ak)|^4. \tag{2.31}$$

By using $b_k^\dagger b_k = n_k$, one can write Eq. (2.18) as the final Bose-Hubbard Hamiltonian

$$H = -\sum_{|k-k'|=1} J_{kk'} b_k^\dagger b_{k'} + \sum_k \left(V_k n_k + \frac{U_k}{2} n_k(n_k-1)\right). \tag{2.32}$$

Here the size of the optical lattice does not necessary have to be infinite, as it is the case with a sine potential. If M is the number of external potential wells, a Fock state can be

defined by a tuple of M occupation numbers n_1 through n_M. So the sums in Eq. (2.32) have a lower bound of 1 and an upper bound of M,

$$H = -\sum_{\substack{k,\,k'\,=\,1 \\ |k\,-\,k'|\,=\,1}}^{M} J_{kk'} b_k^\dagger b_{k'} + \sum_{k=1}^{M} \left(V_k n_k + \frac{U_k}{2} n_k (n_k - 1) \right). \tag{2.33}$$

This Hamiltonian conserves the total number of particles, so the Hilbert space can be reduced to the space of states, where the total number of particles $N = \sum_k n_k$ is constant. This only works if the initial state of the system also lies entirely in this Hilbert space, i.e. the initial state is an eigenstate of operator N.

In principle the coefficients $J_{kk'}$, V_k and U_k can be time-dependent, which leads to a general expression for the Bose-Hubbard model,

$$H = -\sum_{\substack{k,\,k'\,=\,1 \\ |k\,-\,k'|\,=\,1}}^{M} J_{kk'}(t) b_k^\dagger b_{k'} + \sum_{k=1}^{M} \left(V_k(t) n_k + \frac{U_k(t)}{2} n_k (n_k - 1) \right). \tag{2.34}$$

Before proceeding further and solving the time evolution of a system dictated by the Hamiltonian (2.34) numerically, the main numerical tool CUDA [11] must be introduced first. Using CUDA, the algorithms described in chapter 4 can then be parallelized and performed on a graphics card.

3. CUDA and cuSPARSE

This chapter gives an overview of the GPU environment and CUDA, based on the programming guide provided by Nvidia [11].

CUDA (Compute Unified Device Architecture), which is developed by Nvidia, provides an interface for using a GPU as a general purpose computational device. It only supports Nvidia graphics cards and includes an own programming language CUDA-C with the compiler NVCC, which allows to mix CPU (host) and GPU (device) code. CUDA-C is an extension of the ordinary C/C++ language, which contains expressions for memory management, synchronization and marking of host and device scopes.

Similar languages for GPU programming are OpenCL and HLSL (High Level Shader Language) combined with DirectCompute. Using those libraries, one cannot mix host and device code, as device code has to be compiled with a separate compiler as host code. Although both support AMD as well as Nvidia graphics cards, OpenCL runs slower than CUDA [12] and both HLSL and OpenCL device code and the corresponding host code is harder to write, because both are mainly low-level.

It is also possible to use an FPGA (Field Programmable Gate Array) to convert a parallelized algorithm to hardware, which one could design in such a way that the full potential of parallelization is exploited. The downsides of this approach is that programming an FPGA is as low level as it can be, especially communication with other devices connected to the FPGA such as a RAM or a CPU, so debugging and programming are difficult. Moreover FPGAs are clocked at relatively low frequencies, typically at 100 to 200 MHz, so the benefit of maximal parallelization can be invalidated.

Due to these points, CUDA was chosen to implement the algorithms described in chapter 4. The following sections detail how one addresses the GPU interface provided by CUDA.

3.1. The platform: graphics cards

Graphics cards were specifically designed to handle graphics related tasks efficiently. These tasks are highly parallelizable. The pixel colours on a screen are for example independent from each other, as the computation of the colour only depends on the position of the pixel. This is the reason why graphics cards have many more computational

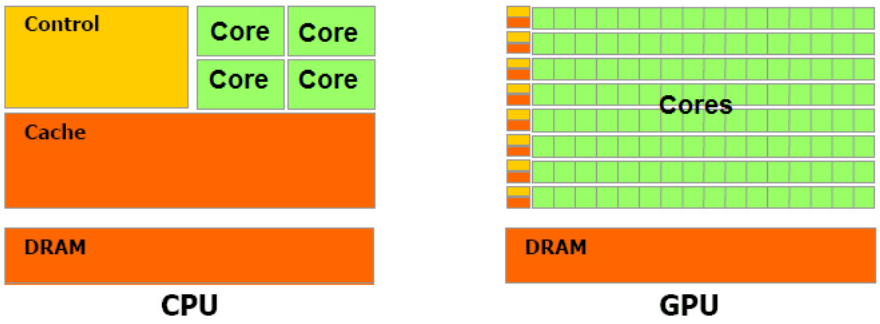


Figure 3.1.: Comparison of typical CPU and GPU chip space occupation taken from [11]. The orange boxes represent on-chip cache and the yellow ones are control logic.

units called cores. These cores also occupy a lot more space on the GPU chip than the cores of a CPU, which is shown in figure 3.1. This means there is less space for cache, memory management and control of the GPU cores, as there is more emphasis on the actual computation of data. GPUs are also optimized for 32-bit floating-point operations, as these execute in mostly only one clock cycle in contrast to CPUs, where these operations take usually more than one clock cycle to complete. This is a huge bonus for scientific computations, as these rely mostly on floating-point operations. Since they require higher accuracy, namely 64-bit floats, most modern GPUs, especially the ones who are designed for scientific tasks, support double precision. So GPUs are suited for highly parallelizable problems and computational intensive tasks.

3.2. Threads

Threads are sequential executed parts of processes on a single core, therefore multiple threads can be performed on multiple cores in parallel. On the GPU, threads can be carried out in groups called blocks with user defined sizes. The threads in a block can share data through a common memory space, called shared memory, which will be discussed in section 3.3.5. Using this memory the user can implement dependencies between threads inside a block.

If the user excludes every dependency between threads, their execution on a GPU is still not independent from each other. As stated above, control space for the cores is reduced on GPU chips, so there is not enough capacity to send operation codes to every single core separately. The solution is to group cores into so-called warps, usually containing 32 or 64 cores, that execute the same operation on different threads simultaneously. This computational architecture is therefore called Single-Command-Multiple-Threads

(SIMT), which is used in almost all graphics card chips. Due to this architecture, problems arise when trying to execute branching operations like an 'if'-operation. If the outcome of a branching operation is different for threads in a warp, the operations after the branch should not be the same for all threads inside this warp. To handle such diverging branches the whole warp just executes all necessary branches sequentially and the individual threads discard changes in the memory of branches which should not be visited according to the outcome of the branching operation.

Consider a situation where an if-else statement is present with a trivial computational task executing when the given expression is true. Otherwise a demanding computational task is carried out. If both results occur inside threads of a warp, both branches are executed on all threads inside this warp. This is even more extreme, if the code has nested diverging branches, which can slow down the computation on the GPU significantly. This concludes, that algorithms with minimal dynamic branching are likely to be faster when executed on the GPU. However, in case that the if-statement can consistently skip computational intensive parts on all threads of a warp, it can actually increase the performance.
An example is shown below on how to define code that is executable by a GPU, called kernel.

Listing 3.1: Definition of a kernel.

```
1 __global__ void add(double* V, double B)
2 {
3     int i = blockDim.x * blockIdx.x + threadIdx.x;
4     V[i] = V[i] + B;
5 }
```

A function can be marked as a kernel by the `__global__` specifier. Kernels cannot have a return value, so they are defined as `void` functions. Pointers passed to kernels have to point to GPU global memory addresses when no other memory type is specified.

To get a unique ID for every thread, one has to use the intrinsic variables `blockDim`, `blockIdx` and `threadIdx`. `blockDim` specifies the number of threads per block. `blockIdx` is an integer that runs from 0 to `blockDim` - 1, so it is the same value for all threads inside one block. `threadIdx` numbers the individual threads inside a block. The variable `gridDim` is also provided, which is assigned to the total number of blocks executing the current kernel. These variables are automatically set, if the program is inside the scope of a kernel. All of them have 3 components `x`, `y` and `z`. In the kernel example 3.1 only `x` is used, as it corresponds to the dimensionality of the problem of iterating over a 1D array. In general it is beneficial to reflect the dimensionality of a given problem in the thread enumeration, as caching of data takes the locality of accesses from different 'nearby' threads into account, especially texture memory accesses (see section 3.3.2). The entirety of all blocks that have to be executed is known as a grid. The whole thread enumeration hierarchy is shown in figure 3.2 for a 2 dimensional grid.

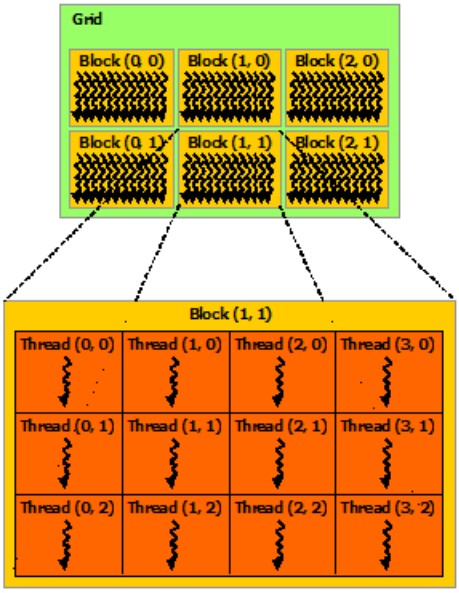


Figure 3.2.: Thread enumeration inside a 2 dimensional grid taken from [11].

For mathematical computations CUDA provides intrinsic functions like `dot`, `cross` or `cos` that have the potential to utilize hardware structures, which are often several orders of magnitude faster than any software implementations.

The kernel in listing 3.1 takes a double array `V`, adds a variable `B` to every component of the array and writes the result into the same array. Note that a single thread only processes one element of the input array, whereas the entirety of all threads can process the whole array `V`. The host code in listing 3.2 shows how this kernel can be executed, together with the proper allocation and readout of the data which will be explained in the following subsection 3.3.

Listing 3.2: Execution of a kernel.

```
1  int size = 10;
2  //initialize the array with data
3  double* V = new double[size];
4  for(int i = 0; i < size; ++i)
5      V[i] = (double)i * 0.5;
6
7  //copy the data to the GPU
8  double* device_V;
9  cudaMalloc((void **)&device_V, size * sizeof(double));
10 cudaMemcpy(device_V, V, size * sizeof(double),
11         cudaMemcpyHostToDevice);
12 //execute kernel, which will add 2 to the components of the array
13 add<<<size,1>>>(device_V, 2.0);
```

```
14 //copy the results from the GPU
15 cudaMemcpy(V, device_V, size * sizeof(double),
16         cudaMemcpyDeviceToHost);
17
18 //print out resulting array
19 for(int i = 0; i < size; ++i)
20     std::cout << "Value␣of␣the␣resulting␣array␣at␣" << i
21         << "␣is␣" << V[i] << std::endl;
22 //clear the memory
23 cudaFree(device_V);
24 delete[] V;
```

The kernel is called just like any ordinary function, except that one has to specify two extra parameters in `<<<...>>>` brackets. This call is seen in the 13th line of listing 3.2. The first parameter corresponds to the number of blocks and the second one defines the number of threads per block. It is possible to add an optional third parameter to the bracket, that describes how much dynamic shared memory is needed per block. Shared memory is addressed in detail in section 3.3.5. The total number of threads this call starts on the GPU is `blockDim.x * gridDim.x` for 1 dimension, which is the grid size. It is notable that these sizes cannot be arbitrarily large. For example the maximal amount of threads per block is limited to 1024 on current graphics cards. In the above program the number of blocks is set to `size` and the amount of threads per block is 1, so the total amount of threads is also `size`. That means that the kernel will run over all elements of the array, because it has the same amount of elements as threads.

3.3. Memory management

GPUs have little chip space dedicated to memory management, so the user is in charge of handling memory transactions and using the correct memory for a given task.

To allow for a higher bandwidth between the main memory and the GPU, modern graphics cards have their own RAM separated from the host one. This device memory is accessible by the host via copy operations. In addition to the main memory the GPU also has specialized memory like constant, shared and texture memory. The entire memory hierarchy and possible connections between them is shown in figure 3.3.

3.3.1. Global memory

The global memory can be accessed by all threads. This memory type is cached, which leads to shorter access times for memory locations frequently visited by the cores. It takes many clock cycles to execute an uncached read or write operation on global memory, as

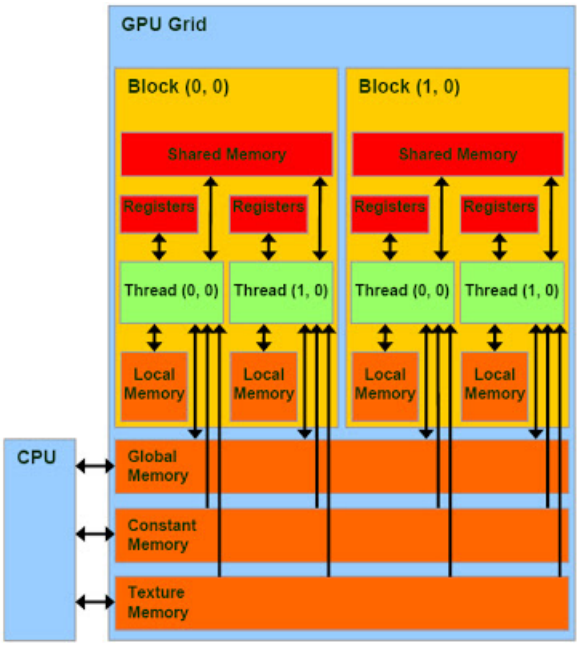


Figure 3.3.: CUDA memory types and their relations to each other and the executing threads taken from [13]. Possible host connections to GPU memory spaces are also shown.

every access must be serialized and gathered across multiple threads. This is especially true for random access patterns to the memory. Although the GPU can hide latencies by executing other threads first on the waiting core, one should nevertheless use global memory only when necessary. To copy data that has to be processed to the device, one firstly has to reserve enough space to hold the data on the GPU. An example code for allocating memory on the device is shown below.

Listing 3.3: Allocation of a double array of size 10 on the GPU's global memory.

```
1 int size = 10;
2 double* device_array;
3 cudaMalloc((void **)&device_array, size * sizeof(double));
```

The cast to `void **` is necessary, because the function `cudaMalloc` has no return values and compatibility for all data types must be present. This code is comparable to the allocation of an ordinary array of same size in the host code below.

Listing 3.4: Allocation of an double array of size 10 accessible only by the CPU.

```
1 int size = 10;
2 double* host_array = (double*)malloc(size*sizeof(double));
```

The output of the function `cudaMalloc` is a pointer `device_array` to the local virtual address of the allocated device memory. Local means that the address belongs to the

GPU global memory space. To insert data into this memory, a host array containing the data has to exist. This array can then be copied to the location returned by the before mentioned function `cudaMalloc`. This can be done with the `cudaMemcpy` function.

Listing 3.5: Transfer of data from a host array to the GPU global memory.

```
1 cudaMemcpy(device_array, host_array, size * sizeof(double),
2         cudaMemcpyHostToDevice);
```

The first two arguments of this function are destination and source respectively. The third argument specifies the size of the array in bytes and the last one the direction of the copy process. In the example in listing 3.5 it is set to `cudaMemcpyHostToDevice`, so `device_array` must live on GPU global memory space and `host_array` on host space. Other copy operations are also feasible, like `cudaMemcpyDeviceToHost` to retrieve data back from the GPU. `cudaMemcpyDeviceToDevice` is an internal copy process between two GPU memory addresses and `cudaMemcpyHostToHost` is equivalent to a `memcpy` of host data.

3.3.2. Texture memory

Accesses to the texture memory are read-only and cached. Thereby the cache exploits the spatial locality of accesses of nearby threads, so if a thread reads at one position in the texture, it is likely that another thread with a similar thread-ID reads data near that position. These positions can have either 1, 2 or 3 dimensions. There are also special texture memory accesses that incorporate interpolation between adjacent data. So texture memory is useful when dealing with continuous data in 1D, 2D or 3D space, like air pressure, an electric field or 2D textures for graphical applications, which will be accessed by the threads in a spacial ordered pattern.

3.3.3. Constant memory

Constant memory can only be set by the host. One declares a variable to be part of constant memory by marking it with `__constant__`. Constant memory can be as fast as one clock cycle to read, if all threads inside a half warp read at the same memory address. If not, the reads get serialized. The maximal amount of constant memory is usually 64 kB.

3.3.4. Local memory

Local memory behaves much like global memory. The main difference to global memory is that threads cannot access local memory of other threads. If arrays or variables exceed

the number of registers a core has, it is automatically allocated in local memory and accessed from there. One can force a variable to be located inside the local memory of the threads by assigning it the `__local__` specifier.

3.3.5. Shared memory

If one wants to include a dependency between multiple threads, one has to rely on shared memory. When starting threads with some program to execute, the user has the possibility to start these threads in groups called blocks. These blocks should not be confused with warps. Warps are hardware dependent limitations on the way operations are sent to the cores, whereas blocks are collections of threads which share common on-chip memory, i.e. shared memory. This memory cannot be accessed by the host. The shared memory location is actually the same as the location of the cache, so one can configure the GPU to use less cache and more shared memory or vice versa.

To mark a variable as shared memory, one has to specify the type of the variable as `__shared__`. When using one common memory across multiple threads, problems due to synchronization arise, like race conditions between threads. To overcome this, there is a special intrinsic function `__synchthreads()`. Threads will halt at this function until all other threads inside the block arrive at the `__synchthreads()` statement. If the block size is smaller than the warp size, one does not have to use this function, as all threads inside a warp are already synchronized. The benefit of using shared memory over global memory for inter-block communication is due to the fact, that shared memory is located on the same chip as the cores. This means that it is much faster to access the data stored inside that memory type. Actually it can be as fast as one clock cycle to read and write to shared memory if all threads access different memory banks at the same time. Memory banks are chunks of shared memory that, if accessed at the same time by multiple threads, results in bank conflicts. If this happens the data is sent serialized to all threads, so if for example two threads access the same memory bank, the time to acquire the data is doubled. The data inside an array is spread in sequential order across the memory banks in 32-bit slices, so access to an array consisting out of 32-bit elements at `threadIdx.x` should result in as little bank conflicts as possible. If the block and array size is the same as the number of memory banks, usually 16 or 32, there should be no bank conflicts.

3.4. cuSPARSE

CUDA does not only provide the basic API, it also delivers some useful libraries containing algorithms, that run on the GPU. Specifically the cuSPARSE library [14] is used to compare the performance of the algorithm described later to the well optimized algorithms for sparse matrix computations. More precisely, the implementation of the cuSPARSE matrix-vector multiplication is applied as an alternative to the algorithm illustrated in chapter 4, which also operates on a sparse matrix though without actually storing the matrix in memory.

4. Implementation

To determine how the state of a system evolves one must solve the time-dependent Schrödinger equation

$$H\left|\psi\right\rangle = i\hbar\frac{\partial}{\partial t}\left|\psi\right\rangle . \tag{4.1}$$

For the Bose-Hubbard model the Hamiltonian is given by Eq. (2.34). At regimes of real experiments, where particle numbers and site counts are high, the Hamiltonian cannot be analytically diagonalized anymore. Numerical methods are not yet powerful enough to investigate these regimes, but they can approximate them, either through extrapolation to higher numbers of particles and sites or through approximations on the methods themselves. These include the mean field approximation [15] and the Bogoliubov-Born-Green-Kirkwood-Yvon hierarchy, also known as Bogoliubov Backreaction method [16, 17] when it is combined with the Bose-Hubbard model. This chapter discusses an implementation of an exact approach to solve the time-evolution of Eq. (2.34). This implementation will be highly optimized for execution on a GPU, which will be shown in chapter 5.

4.1. Conversion of the problem to a linear algebra one

The state $|\psi\rangle$ can be expressed as a superposition of Fock states $|n_1, ..., n_M\rangle$,

$$\left|\psi\right\rangle = \sum_{n_1+n_2+...+n_M=N} \underbrace{\left\langle n_1, ..., n_M|\psi\right\rangle}_{C_{n_1,...,n_M}} \left|n_1, ..., n_M\right\rangle . \tag{4.2}$$

So $|\psi\rangle$ can be represented as a vector consisting of the amplitudes $C_{n_1,...,n_M}$,

$$\left|\psi\right\rangle \equiv \vec{y} = \begin{pmatrix} C_{n_1^1,...,n_M^1} \\ C_{n_1^2,...,n_M^2} \\ ... \end{pmatrix} . \tag{4.3}$$

In this representation the Hamiltonian (2.34) becomes a matrix,

$$H^{n_1^h,...,n_M^h}_{n_1^g,...,n_M^g} = \langle n_1^g, ..., n_M^g | H | n_1^h, ..., n_M^h \rangle \tag{4.4a}$$

$$\implies H \equiv \begin{pmatrix} H^{n_1^1,...,n_M^1}_{n_1^1,...,n_M^1} & H^{n_1^2,...,n_M^2}_{n_1^1,...,n_M^1} & ... \\ H^{n_1^1,...,n_M^1}_{n_1^2,...,n_M^2} & H^{n_1^2,...,n_M^2}_{n_1^2,...,n_M^2} & ... \\ ... & ... & ... \end{pmatrix}. \tag{4.4b}$$

The order in which the tuples of occupation numbers $(n_1^g, ..., n_M^g)$ appear in the vector (4.3) can be in principle arbitrary. The only restriction is that every combination of $(n_1^g, ..., n_M^g)$ with $N = \sum_k n_k$ must be present inside the vector (4.3). So the Schrödinger equation (4.1) becomes a system of coupled differential equations of the form

$$\dot{\vec{y}} = A \cdot \vec{y}, \tag{4.5}$$

where $A = -iH/\hbar$. $\vec{y}$ can be integrated by means of standard numerical methods like the Runge-Kutta method, which consists of only matrix-vector multiplications and additions. If one is interested in the evolution of $\vec{y}$ with time-independent potentials, the matrix can be just precomputed and stored. By using cuSPARSE from section 3.4 the necessary matrix-vector multiplications can be executed on the GPU.

If there are no time dependencies of the potentials, it is also possible to integrate Eq. (4.5) by applying an approximation of the time-evolution operator $U(\Delta t)$ for small Δt repetitively to the state. The Taylor expansion of the time-evolution operator up to the order of $\mathcal{O}(\Delta t^3)$ is shown below,

$$\begin{aligned} U(\Delta t)\vec{y} &= \exp\left(\frac{-iH\Delta t}{\hbar}\right)\vec{y} \\ &= \exp(A\Delta t)\vec{y} \\ &= \sum_{n=0}^{\infty} \frac{(A\Delta t)^n}{n!} y \qquad (4.6a) \\ &\approx \vec{y} + \Delta t A \cdot \vec{y} + \frac{1}{2}\Delta t^2 A(A \cdot \vec{y}) + \mathcal{O}(\Delta t^3). \qquad (4.6b) \end{aligned}$$

With time dependencies inside the Hamiltonian, this method cannot be applied. In this case, the matrix has to be modified in every time step. This would result in an application purely limited by the bandwidth of the GPU memory. This is not optimal, as the GPU is designed for computational intensive tasks.

4.2. Matrix-free matrix-vector product

As the time evolution requires only matrix-vector multiplication, the matrix does not necessarily have to be stored explicitly. One only needs a function that takes the current

state-vector and an index i, to indicate which state-amplitude this function has to return. By iterating over all non-zero matrix elements of the i-th row of the matrix A, multiplying those with the corresponding amplitude found in the current state-vector and adding them together, one gets the result for the i-th value of the matrix-vector product. The method described here is called a matrix-free method, as it does not require storing the matrix in memory to perform a matrix-vector product. This results in a much better utilization of the GPUs resources, as there is less time wasted waiting for the matrix data to arrive. A matrix-free representation of a matrix-vector product is also useful to find eigenvalues and states using iterative methods like the Lanczos algorithm [18].

It is now only a matter of finding these non-zero matrix elements for a given i. Consider the second term of the Bose-Hubbard Hamiltonian (2.34). It consists only of n_k operators with the property

$$\hat{n}_k \left|n_1, ..., n_k, ..., n_M\right\rangle = n_k \left|n_1, ..., n_k, ..., n_M\right\rangle . \tag{4.7}$$

The action of $\hat{n}_k$ does not change the state itself, hence the index of the state i does not change either. The problem now is to find n_k, where the only information given is the index of the state. That means that one needs a function of the kind

$$O : \mathbb{N} \to (\underbrace{\mathbb{N}, ..., \mathbb{N}}_{M \text{ times}}) = i \to (n_1, ..., n_M). \tag{4.8}$$

This in fact reflects just the enumeration of the basis-vectors in Eq. (4.3). This function can be precomputed and then fed as a look-up table to the matrix-vector multiplication.

Until now the order of the Fock state amplitudes inside the vector (4.3) was not important. However, it becomes important when evaluating the first term of Eq. (2.34). The reason behind that is, that the action of $b_k^\dagger b_{k'}$ transforms the initial state $\left|n_1, ..., n_M\right\rangle$ to another Fock state, i.e.

$$b_{k+1}^\dagger b_k \left|n_1, ..., n_k, n_{k+1}, ..., n_M\right\rangle = \sqrt{(n_{k+1}+1)n_k} \left|n_1, ..., n_k - 1, n_{k+1} + 1, ..., n_M\right\rangle . \tag{4.9}$$

As a consequence the index of the vector entry with whom $\sqrt{(n_{k+1}+1)n_k}$ has to be multiplied is not equal to the initial index i. With k and Eq. (4.8) one can find the resulting tuple of occupation numbers $(n_1, ..., n_k - 1, n_{k+1} + 1, ..., n_m)$. The only thing missing is the inverse mapping of Eq. (4.8). A trivial way of calculating that inverse function, is to precompute it into an array of size N^M, so the time of the resulting look-up is $\mathcal{O}(1)$. However, there is a major problem with this approach: with large N and M most of the entries in this look-up table would not be assigned, as these entries return indices which correspond to $N \neq \sum_k n_k$. So this method would waste too much memory to be practical.

Another approach to invert the function (4.8) is described in [19]. J. M. Zhang and R. X. Dong assign to every basis-vector a hash value. The order of the basis-vector is then

defined by sorting the corresponding hash values by magnitude. The ordered list of hash values is stored in an array. If one wants to find the index of a tuple of occupation numbers $(n_1, ..., n_m)$, one must first calculate its hash value. Performing a search of this hash value in the sorted array of all hash-values yields the position of the hash value, which is the index one is looking for. The downside of this approach is the search operation, as the access to the state vector does not longer have a time complexity of $\mathcal{O}(1)$ but instead one of $\mathcal{O}(\log L)$ with L being the size of the search space. It also requires multiple random accesses to the global memory, which cannot be cached efficiently. This leads to decreased performance, as the inverse of the mapping (4.8) will be calculated multiple times, each with multiple accesses to global memory. A search operation also requires divergent branching, which reduces the performance of this method. The comparison of this method and the following is measured and discussed in section 5.4.

The method of choice is an algebraic representation of the lexicographic order [20] of occupation numbers, defined as

$$(a_1, b_1) < (a_2, b_2) \equiv (a_1 < a_2) \vee ((a_1 = a_2) \wedge (b_1 < b_2)). \tag{4.10}$$

Here a_1, a_2, b_1, b_2 are elements of a set with a specified order defined with $<$. a_1, a_2, b_1, b_2 can be part of natural numbers or itself a tuple, where the order is defined by Eq. (4.10). By using $(a, b, c) \equiv (a, (b, c))$, one can recursively construct the order for tuples of natural numbers $(\mathbb{N}, ..., \mathbb{N})$. Then the index i of the smallest tuple is 1 and the next smallest is 2 and so forth. For $N = 2$ and $M = 3$ this order is shown in table 4.1.

To determine the dimension of the Hilbert space, i.e. the number of different occupation number tuples, one can represent these tuples in another way, where the occupation number tuples are filled from one side to the other with 'particles'. The accumulation is then represented by another tuple consisting of 1 and 0, which must be read sequentially to retrieve the occupation numbers. One starts at the first potential well. 1 means a 'particle' is dropped in the current site. When a 0 occurs, the next site gets filled. An example of this representation is displayed in table 4.1, where it is named 'Particle representation'. The resulting tuple contains N times 1 and $M - 1$ times 0. Using

Occ. number tuple	Particle representation	Index
$(0, 0, 2)$	$(0, 0, 1, 1)$	1
$(0, 1, 1)$	$(0, 1, 0, 1)$	2
$(0, 2, 0)$	$(0, 1, 1, 0)$	3
$(1, 0, 1)$	$(1, 0, 0, 1)$	4
$(1, 1, 0)$	$(1, 0, 1, 0)$	5
$(2, 0, 0)$	$(1, 1, 0, 0)$	6

Table 4.1.: Order of the occupation number tuples for $N = 2$ and $M = 3$

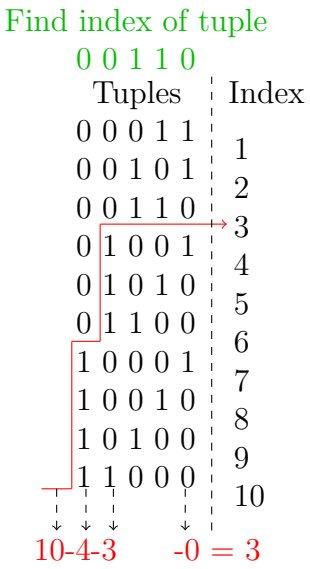


Figure 4.1.: Index calculation of a $(0, 0, 1, 1, 0)$ tuple by subtraction of tuple sub-set sizes. The starting point is the total size of the Hilbert space.

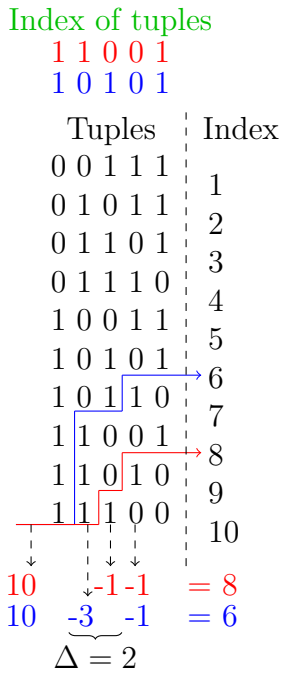


Figure 4.2.: Demonstration, that the index difference of tuples, which are similar except for an exchange of a pair of adjacent 0, 1, can be calculated with only two accesses to $D(N, M)$ and one when using relation (4.15).

combinatorics, one finds that the total number of arrangements is

$$D(N, M) = \frac{(N + M - 1)!}{N!(M - 1)!}, \tag{4.11}$$

which corresponds to the dimension of the Hilbert space. The index can now be computed by the algorithm A.1 (all following listings can be found in the appendix), which skips tuples with a different beginning as the tuple whose index must be calculated. This is achieved by accumulating the dimensions of the set of sub-tuples, which consist of the remaining 1 and 0. This skip is only performed if the current value of the tuple is 0, as the current index is pointing to a tuple with a 1 at the current position. Otherwise the index does not have to change, as beginning of the given tuple and the tuple at index i are identical. A practical example of an application of the algorithm is illustrated in figure 4.1.

The algorithm described in listing A.1 can be rewritten to take an array of the occupation

numbers, so it is not necessary to convert the occupation number tuple to the particle representation. Now one has everything necessary, to compute Eq. (4.9) algorithmically, but there is still a problem. The computation of $D(N, M)$ can slow down the execution of the matrix-vector multiplication, as the function `getIndex` get invoked often. To calculate $D(N, M)$ one has to evaluate 3 factorials, which are time consuming. The solution is to precompute $D(N, M)$ for all possible combinations of N and M. There is an even better solution: consider how $b^\dagger_{k+1} b_k$ changes an arbitrary tuple in the particle notation. For example $(1, 1, 0, 0)$ and $b^\dagger_2 b_1$ results in a particle representation tuple $(1, 0, 1, 0)$. By examining this behaviour one can work out that a transition of a particle to a neighbouring site is just an exchange of an adjacent pair of 0 and 1 in the particle representation tuple. It is now possible, to calculate the difference Δ between the original index and the index of the resulting tuple with only one $D(N, M)$ accesses, because the dimensions of the sub-sets before and after the pair of 0 and 1 which get exchanged remains the same in the calculation of both indices. This is the case, as the amount of remaining particles and sites at $k - 1$ and $k + 2$ does not change. This behaviour is shown in figure 4.2 for a transition between a $(1, 1, 0, 0, 1)$ and a $(1, 0, 1, 0, 1)$ tuple.

To compute Δ, one has to determine the value which gets subtracted at the position of the 0, which is involved in the exchange of a particle, inside the original and the resulting tuple. For the tuple with the configuration $C = (a_1, ..., a_{p-2}, 1, 0, b_1, ..., b_m)$, the index has to skip over all tuples $(a_1, ..., a_{p-2}, 1, 1, c_1, ..., c_m)$, because these tuples do not contain the wanted tuple C. The size of the skip is then just the number of possible configurations of $(a_1, ..., a_{p-2}, 1, 1, c_1, ..., c_m)$, where the numbers $a_1, ..., a_{p-2}$ are fixed. This means one has to only calculate the total amount of possible sub-tuple configurations $(c_1, ..., c_m)$. If the number of 1 and 0 in $(b_1, ..., b_m)$ are known, referred as #remaining particles and #remaining sites $- 1$ respectively, the number of 1 and 0 in $(c_1, ..., c_m)$ are #remaining particles $- 1$ and #remaining sites. Using this, the skip size is

$$D(\#\text{remaining particles} - 1, \#\text{remaining sites}). \tag{4.12}$$

The only difference in a tuple configuration $(..., 0, 1, ...)$ is that the 0 precedes 1, so there is an additional remaining particle after the exchange of the 0. This concludes that one receives a subtraction of

$$D(\#\text{remaining particles}, \#\text{remaining sites}) \tag{4.13}$$

at position $p - 1$ in the index calculation of the tuple. Then the difference between the indices of these two tuples is

$$\begin{aligned} \Delta =& D(\#\text{remaining particles}, \#\text{remaining sites}) \\ & - D(\#\text{remaining particles} - 1, \#\text{remaining sites}). \end{aligned} \tag{4.14}$$

Using the relation

$$\binom{n}{k} - \binom{n-1}{k} = \binom{n-1}{k-1}, \tag{4.15}$$

Δ can be simplified to

$$\Delta = D(\#\text{remaining particles} - 1, \#\text{remaining sites} - 1). \tag{4.16}$$

The corresponding algorithm for the calculation of Δ can be found in listing A.2. To note here is that it is possible to calculate arbitrary transitions through multiple application of this skip method. The function can also be written to take an occupation number tuple. To make the algorithm even faster, one can precalculate Δ in a 2D-array which contains the according Δ-values for all possible combinations of remaining particles and remaining sites.

Now one has everything necessary, to write down the algorithm for the matrix-vector multiplication with a matrix described by Eq. (2.34). This algorithm is shown in listing A.3.

The system parameters like the tunnelling rates $J_{kk'}$ have to be computed beforehand and stored in arrays. By doing so the eventual time dependency does not have to be calculated on the fly during the matrix-vector multiplication. These arrays have to be copied to the GPU, which costs execution time, however, this latency is insignificant in contrast to the execution time of the algorithm itself. Moreover, if the system parameters are calculated on the fly, even simple functions like an oscillation will slow down the computation, because these functions get invoked often during the matrix-vector multiplication.

4.3. Optimizations for the GPU enviroment

As previously stated, precomputed values of computations are not always adequate due to the architecture of the GPU. Here the heavy use of look-up tables in algorithm A.3 is justified, as the accesses result in a lower overhead than calculating them on the fly. The reason is that either these accesses are in a sequential order inside a warp, an example are accesses to `occupation_numbers_l`, or the look-up tables are small enough to fit inside the on-chip GPU cache, which has usually a size of 48 kB. The latter case holds for example for `Delta_l`. Due to that, all look-up tables with random access patterns like `Delta_l` and `SQRT_l` reside inside the global memory.

To reduce the bandwidth used by the algorithm, one can pack `occupation_numbers_l` more densely by assigning every occupation number 8 or even 4 bits, as it is unlikely that one needs to simulate more than 255 particles.[1] The main reason for that is the exponential increase in storage necessary for the state vector, which is shown in table 4.2. It is also beneficial to load the necessary `occupation_numbers_l` values inside an array in sequence, so that accesses to global memory can be serialized. For the same reason the system parameters J, V and U should be written to a shared memory array at the

[1] Although it is possible to use more bits per occupation number if it is absolutely mandatory.

M	D(255,M)	Memory size of the state vector
1	1	16 bytes
2	256	4 kB
3	32896	526 kB
4	2829056	45.3 MB
5	183181376	2.9 GB
6	9525431552	152 GB

Table 4.2.: Double precision state vector dimensions and memory consumption for $N = 255$ and different M. Every element uses $2 \cdot 8$ bytes, as the state amplitudes are complex double-precision values. It is clear to see that after $N = 5$, it is not feasible, to keep the entire state vector on the GPU memory.

start of the kernel, as these are used in all threads inside a warp and are accessed at the same time with the same index. In this situation, the shared memory is broadcasted to all threads, which takes the same amount of time as an access to a register. It is also possible to write them into registers, but this would lower the number of concurrent threads, because more registers are needed per thread.

Another way of optimizing the algorithm is to execute multiple rows of the matrix-vector multiplication inside a kernel. Although it is possible using algorithm A.3 to calculate every element inside a separate thread in parallel, modern GPUs cannot execute such an amount of threads simultaneously, which are required if N and M are large. Sequential code is needed to reduce the parallelism so that it fits the capability of the hardware. As every launch of a new thread inside the GPU has a small overhead, it is more convenient to write sequential code inside the kernel than outside. This can be realized by a for-loop which iterates over a range of row indices i.

It is also beneficial to set the block-size to a value dividable by the warp size of the GPU, which is usually 32 or 64. No idle threads are launched that way.

Taking all these considerations into account, one can write down the resulting kernel which is shown in listing A.4. It is using the function defined in listing A.3 with a few modifications to suite CUDA-C.

4.4. Runge-Kutta methods

Two Runge-Kutta methods where implemented to compute the time evolution of the state vector. The first one is the standard 4th order Runge-Kutta method [21]. The second one is described in [22], which is an embedded Runge-Kutta method involving

the evaluation of 4th order and 5th order values simultaneously. This will later help to ensure that the integration error stays within a desired range. Both methods integrate a system of equations

$$\dot{\vec{y}}(t) = \vec{f}(t, \vec{y}) \tag{4.17}$$

using steps of size h. Then t_n and $\vec{y}_n$ denote the corresponding values at step n. Inside these steps, Runge-Kutta methods evaluate the function $\vec{f}$ multiple times with different t and $\vec{y}$. These evaluation positions itself are determined through previous calculations of $\vec{f}$. These relations are shown in the following formulas,

$$\vec{k}_j = \vec{f}\left(t_n + a_i h, \vec{y}_n + h \sum_{h=1}^{j-1} b_{jh} \vec{k}_h\right), \tag{4.18a}$$

$$\vec{y}_{n+1} = \vec{y}_n + \sum_{j=1}^{s} c_j \vec{k}_j. \tag{4.18b}$$

The coefficients a_i, b_{jh} and c_j can be represented compactly using a Butcher tableau. A general one is depicted in table 4.3. For the here used 4th order Runge-Kutta method, a Butcher tableau is shown in table 4.4.

Eqs. (4.18a) and (4.18b) show, that Runge-Kutta methods need not only matrix-vector multiplications, but also sums of vectors and multiplications of a vector with a constant. The addition and multiplication with a constant can be performed on the CPU, although one has to transfer data between GPU and CPU to do so. Also a CPU implementation does not exploit the parallelizability of these operations. Additionally it is inefficient to keep a copy of the state vector on the CPU main memory, as every necessary manipulation of the state vector can be made on the GPU. Even observables like particle densities can be implemented as kernels. So it is appropriate to implement the vector addition and multiplication with a constant inside kernels, where every element is processed in parallel like it is the case with the matrix-vector multiplications. As stated above, the launch of every new kernel and thread carries a small delay, so the performance of the Runge-Kutta integration can be increased by compressing multiple tasks of vector manipulation which

a_1				
a_2	b_{21}			
a_3	b_{31}	b_{32}		
...	...	...	...	
a_s	b_{s1}	...	$b_{s,s-1}$	
	c_1	...	c_{s-1}	c_s

Table 4.3.: A Butcher tableau containing all necessary coefficients of a Runge-Kutta method.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Table 4.4.: Butcher tableau of the classic 4th order Runge-Kutta method.

can run independently from each other inside a single kernel. It is also possible to reuse device memory in the process to store and compute the next evaluation positions of the function $\vec{f}(t, \vec{y})$. Important to know is that if the execution time of a kernel is longer than a few seconds, the kernel could terminate, which is called a kernel timeout. It is possible to disable this behaviour if kernel timeouts occur.

The same procedure of minimizing the amount of kernel calls can be achieved with the embedded Runge-Kutta method. Note that this method uses more memory and also more computation time per step. On the other hand, if one likes to track the error with the 4th order Runge-Kutta method, one has to fall back to step doubling, which takes even more computation time per step than the embedded Runge-Kutta method. Step doubling involves two half sized steps, which are compared to the outcome of a full size step to approximate the error.

The error is then used to either reset the current step if it exceeds a given limit or to estimate the next step size. This way the total duration of the integration using dynamic step size can be actually lower than one with constant step size, as portions of the integration can be skipped if the error is small enough. The function to determine the step size chosen here is a combination of two boundaries. The first one is the standard rule, that the error per step must be smaller than some user defined value ϵ_1. To derive that condition we use the fact that with a 4th order Runge-Kutta method the error scales in the worst case with the power of 5 of the step size. So one can write

$$\frac{\epsilon'}{\epsilon} = \frac{h_{\text{next}}^5}{h_{\text{now}}^5}, \tag{4.19}$$

which represents how the error would change with a change of the step size. By solving this equation for h_{next}, the result is the update rule

$$h_{\text{next1}} = h_{\text{now}} \left| \frac{\epsilon_1}{\epsilon} \right|^{0.2}, \tag{4.20}$$

where ϵ is the current estimated error, h_{now} is the current step size and h_{next1} is the suggested next step size. The second rule is a boundary on the accumulated error ϵ_2 at the end of the integration. This condition that the accumulated error must be smaller than some limit is denoted as

$$\begin{aligned} \epsilon_2 &\geq \epsilon_{\text{accu}}(T) \\ &= \sum_{t_n=t}^{T} \epsilon(t_n) + \epsilon_{\text{accu}}(t) && \text{(4.21a)} \\ &\approx \epsilon' \frac{T - (t + h_{\text{next2}})}{h_{\text{next2}}} + \epsilon_{\text{accu}}(t + h_{\text{next2}}), && \text{(4.21b)} \end{aligned}$$

where T is the total integration time at the end of the simulation and $\epsilon_{\text{accu}}(t)$ is the accumulated error before time t. W.l.o.g. the time of origin is $t_0 = 0$. The sum is

approximated using the assumption that the next error ϵ' remains the same for the rest of the simulation. Eq. (4.21a) can be further approximated by assuming that the accumulated error also follows the next error per step until time $t + h_{\text{next2}}$, which leads to

$$\epsilon_{\text{accu}}(t) \approx \epsilon' \frac{t}{h_{\text{next2}}} \tag{4.22a}$$

$$\implies \epsilon_2 \gtrsim \epsilon' \frac{T}{h_{\text{next2}}}. \tag{4.22b}$$

Now one can simply insert Eq. (4.19) into Eq. (4.22a) and solve for h_{next2}, which results in

$$h_{\text{next2}} \gtrsim \left(\frac{\epsilon_2 h_{\text{now}}^5}{\epsilon T} \right)^{\frac{1}{4}}. \tag{4.23}$$

To achieve the ϵ_2 limit with a minimal amount of steps the right side can be set equal to the left, where the left side has to be multiplied by with a safety factor $0 < S < 1$ so the accumulated error does not overshoot the given limit ϵ_2. So Eq. (4.23) becomes

$$h_{\text{next2}} = S \left(\frac{\epsilon_2 h_{\text{now}}^5}{\epsilon T} \right)^{\frac{1}{4}}. \tag{4.24}$$

Combining the two rules, the next step size is chosen as the minimum of both h_{next1} and h_{next2}. Now one only needs to calculate the current step error ϵ. This error can be approximated by subtracting the resulting vector of a 5th order step from a 4th order one. The result is a vector with an error for every element of the state vector. ϵ is then in the worst case the maximum of all errors in this vector, which can be computed in parallel using CUDA. Firstly every thread inside of a block copies a value from the error vector to the shared memory. A prefix scan [23] is then applied to find the maximum of these errors and write it into another smaller array. This kernel has to be executed repeatedly on the resulting arrays until the error vector is small enough to be processed by the CPU, which then iterates over this last error vector to output the maximum.

5. Benchmarks

In this chapter the algorithm for matrix-vector multiplication will be compared to other methods. These methods include an implementation of the very same algorithm on a CPU and one involving cuSPARSE. A method requiring a hash-function for index computation is also examined [19]. To do so, the timings and memory consumptions of all methods are measured. The timings are determined using CUDA events [11], which incorporate a function `cudaEventElapsedTime` that measures the time between two events, for example one before and one after the execution of GPU kernels. To increase the accuracy of the measurements, multiple matrix-vector multiplications are performed in succession. The duration of the execution of all these kernels is then divided by the number of launched kernels to yield the mean duration of a matrix-vector multiplication. The performance of the CPU implementation was measured using `std::chrono::steady_clock::now()` [24]. Table 5.1 displays the computer hardware used for the performance measurements.

5.1. Performance of the algorithm

In this section the performance of the matrix-vector product described by the algorithm A.3 is measured. A benchmark system with 8 potential wells and variable particle numbers is used. The elements of the input vector are set to random values. The on-site energies, the interaction potentials and the hopping terms are set to a constant value because the computation of these can be arbitrarily complex and the overhead of copying these arrays to GPU memory is negligible small for $M = 8$, as can be seen in figure 5.1. The measurements can be found in figures 5.2 and 5.3. There the maximal amount of particles N is dictated by the size of the device memory.

	Consumer PC	Tesla rack
CPU	Intel Core i5-3470 CPU @ 3.20GHz	Intel Xeon CPU E5506 @ 2.13GHz
GPU	NVIDIA GeForce GT 630	NVIDIA Tesla C2070
RAM	8 GB	48 GB

Table 5.1.: Hardware used for benchmarking.

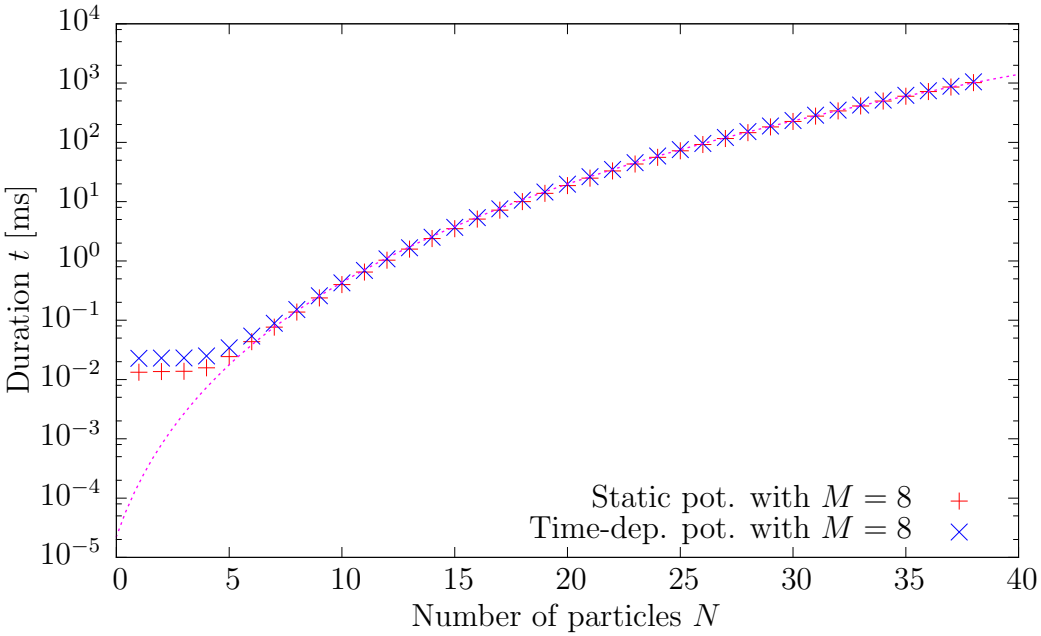


Figure 5.1.: Duration of a matrix-vector multiplication with and without dynamic potential values on a consumer level computer implemented to run on the GPU using algorithm A.3. The one with dynamic potential does not include the actual calculation of the potential terms, just the copying of them to the GPU memory. One can clearly see that the scaling behaviour for large N is the same as the scaling of the state vector dimension. At small N overhead dominates. This overhead is slightly larger when the system parameters are dynamic, because of additional `cudaMemcpy` calls.

Note that the duration of the algorithm scales for large N roughly at the same rate as the number of vector elements which is coupled with the memory consumption. This is as expected, because the for-loop range inside the matrix-vector multiplication in listing A.3 stays constant, as it only depends on the number of sites. The only factor that changes is the number of elements the algorithm has to compute, which is just the dimension of the Hilbert space given in Eq. (4.11). Hence the continuous fit function used in figures 5.1, 5.2 and 5.3 is

$$t(N) = aD(N, M) = a\frac{\Gamma(N + M)}{\Gamma(N + 1)\Gamma(M)} \tag{5.1}$$

with the gamma function $\Gamma(x + 1) = x!$, M set to 8 and a as the fit parameter. Because the performance of Runge-Kutta methods are directly correlated with the speed of the matrix-vector multiplication, which takes the most amount of computation time if the time-dependent computation of J, U and V is excluded, the scaling of the time evolutions step duration follows roughly that of the matrix-vector multiplication.

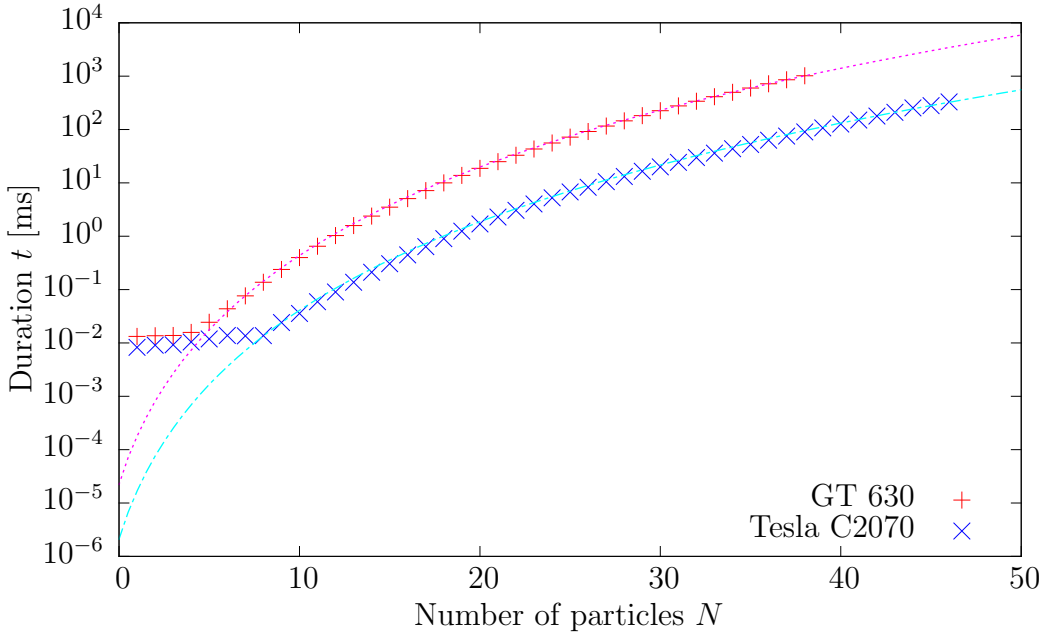


Figure 5.2.: Performance on two different machines for $M = 8$. As expected, the high-end GPU outperforms consumer level hardware.

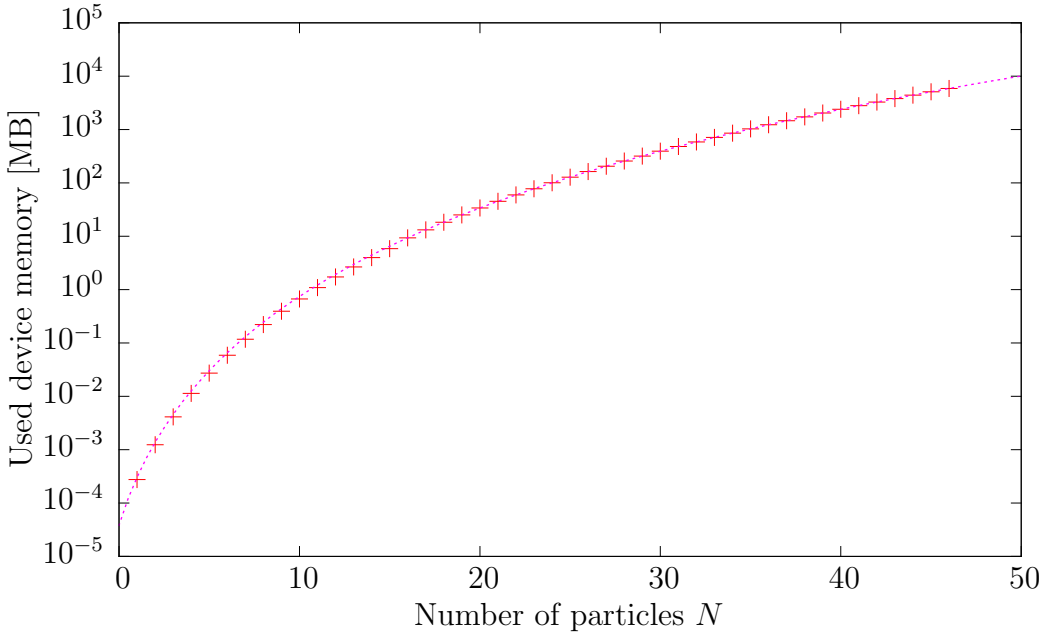


Figure 5.3.: Device memory consumption of the algorithm for $M = 8$. The used memory includes the input state vector and the output array, where the result of the matrix-vector product is stored. Both scale at the same rate as the Hilbert space dimension.

For small N copy time and kernel launch overhead start to dominate, which can be seen in figures 5.1 and 5.2, as the performance curve does not follow the dimension scaling anymore. The late adaption of the Tesla rack measurements to the fit curve compared to the consumer PC ones is due to the similar overhead duration and faster algorithm execution on the Tesla rack.

5.2. GPU versus CPU implementation

The algorithm described in chapter 4 is implemented with C++ and without multi-threading. The results can be seen in figure 5.4 and show that even with consumer hardware a GPU can outperform a CPU with the same task of computing a matrix-vector multiplication on the fly. Figure 5.5 displays how many consumer level CPU cores one professional GPU is worth when executing this algorithm. There

$$\text{Speedup over CPU} = \frac{\text{CPU execution time}}{\text{duration of the GPU implementation}} \tag{5.2}$$

is plotted over the number of particles N.

For large numbers of particles there is a clear performance difference between the CPU and GPU implementation. In fact, the chart shows that a single high-end graphics card can perform the matrix-vector product as fast as 23 consumer level CPU cores in parallel. This proves that it is beneficial to migrate a highly parallelizable algorithm like the one presented in chapter 4 to the GPU. The loss of this high parallelism is seen at low particle numbers, because small particle numbers lower the possible number of concurrent threads. This means that the data transfer and launch time of new kernels get progressively more significant than the actual computational time. This results in the dip of speedup shown in figure 5.5, as the CPU implementation does not suffer from these latencies as much as the GPU does.

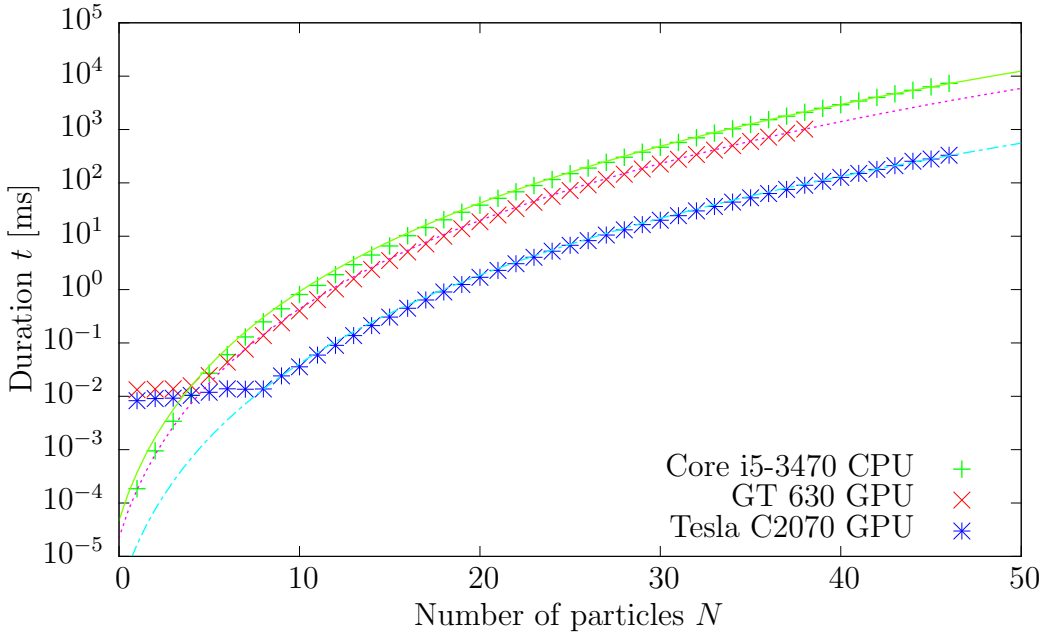


Figure 5.4.: The GPU and CPU performance measurements for $M = 8$.

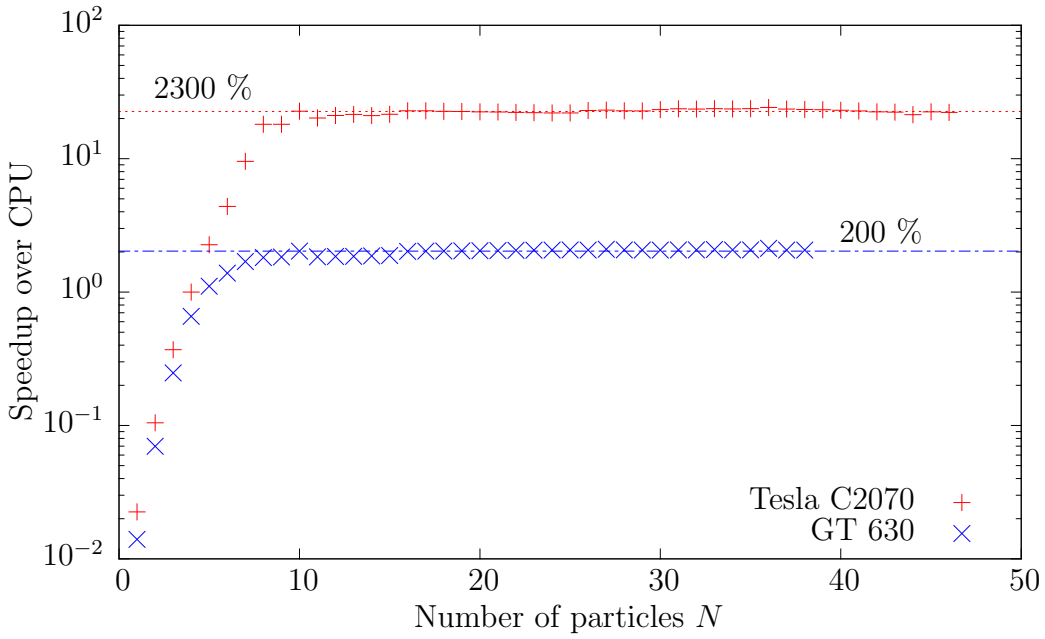


Figure 5.5.: Comparison between a CPU and GPU implementation for $M = 8$. The CPU implementation was carried out single-threaded on a consumer level CPU. The single-threaded CPU performance is then compared to the one of two different GPUs, namely the Nvidia GT 630 and Nvidia Tesla C2070.

5.3. Comparison between a cuSPARSE implementation and the presented algorithm

Here Nvidias well optimized cuSPARSE library implementation of a parallel sparse matrix-vector multiplication is compared with the matrix-free solution derived in chapter 4. The compressed sparse row (CSR) format [14] is used to represent the stored matrix, required by the cuSPARSE implementation. The speedup defined analogously to Eq. (5.2),

$$\text{Speedup} = \frac{\text{cuSPASRE execution time}}{\text{presented algorithm duration}}, \tag{5.3}$$

is shown in figure 5.6.

The speedup at large N is presumably due to the bandwidth constraint of the cuSPARSE matrix-vector multiplication, because there not only the vector but also the matrix has to be accessed from memory. At small N these constraints become insignificant, as the sparse matrices also become small. In fact, at small N cuSPARSE outperforms the presented algorithm, presumably because cuSPARSE does not rely on branching as much as the matrix-free implementation. Because of that it can be assumed that every state transition in algorithm A.3 is calculated even if the site involved in the transition contains no particles. The reason is that it is likely that some thread inside a warp has a

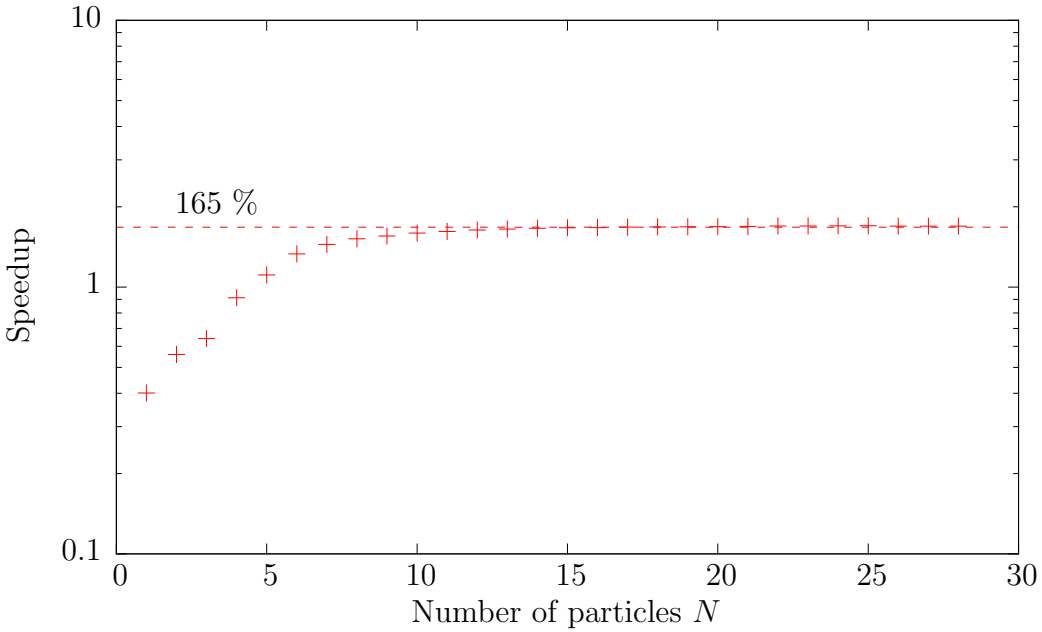


Figure 5.6.: Speedup between cuSPARSE and the matrix-free algorithm on consumer level hardware.

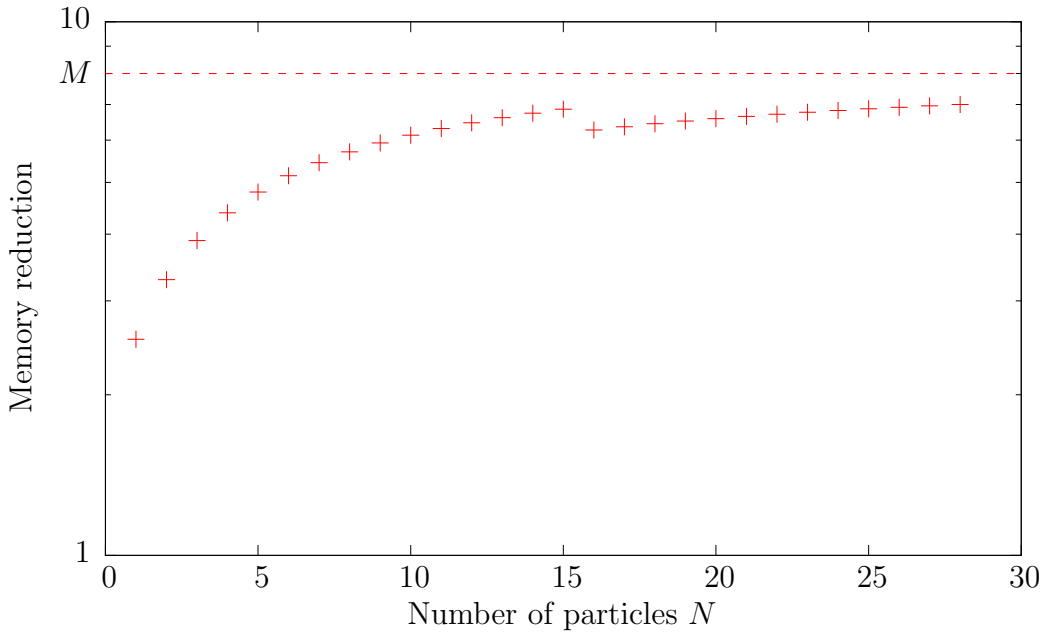


Figure 5.7.: Memory saving when using a matrix-free implementation. The discontinuity at $N = 16$ comes from a switch between 4 to 8 bit occupation number representation, as the maximal number of particles at one site exceeds $2^4 - 1 = 15$.

particle inside this site. Because of the SIMD architecture of the GPU all threads have to calculate this transition. On the other hand cuSPARSE only computes the transitions that actually occur, as the branching is shifted to the creation of the matrix. But with large N this waste of computation time becomes progressively smaller, as the density of particles on each site increases, such that states with empty sites become rare.

To incorporate time-dependent potentials into the cuSPARSE implementation, one would have to rewrite the matrix saved on the device memory at every time step of the integration. This would slow the cuSPARSE implementation down even more.

It is expected that the speedup converges at large N to a constant value, as the time complexity of both algorithms is $\mathcal{O}(MD)$, where D is the size of the state vector, which is the same scaling as the number of non-zero matrix elements that have to be processed. The average number of non-zero elements per row scales linearly with M [19] and there are D matrix rows, so the total number of non-zero matrix elements scales with MD. The access and processing time per non-zero element do not depend on M or N in both cuSPARSE and the presented algorithm, so their time complexities are $\mathcal{O}(MD)$.

The comparison of both memory consumptions is shown in the figure 5.7, where memory reduction is defined as

$$\text{Memory reduction} = \frac{\text{cuSPASRE memory consuption}}{\text{matrix-free memory consuption}}. \tag{5.4}$$

It is also expected that the memory reduction will level out at some constant, as the memory complexity of cuSPARSE is $\mathcal{O}(MD)$ where M is in figure 5.7 constant and the one of the presented algorithm is $\mathcal{O}(D)$. The ratio of both complexities is not exactly M, as eventual extra memory depends on the implementation, like the precomputed occupation numbers, which are taken into account in figure 5.7.

5.4. Comparison of the Bisection Method and the presented algorithm

The performance of the kernel A.4 is compared to a CUDA implementation of a Bose-Hubbard matrix-vector multiplication method described in [19]. There a hash-function assigns every Fock state a unique hash value. A list of all hash values is stored and sorted by their magnitude. The index at which a hash value occurs corresponds to the index of the Fock state which generates this hash value. To find the index of a given Fock state, its hash value is searched in the list of all hash values by using the Bisection algorithm. Such a list of hashes consumes additional memory compared to the presented algorithm. The time complexity of the Bisection Method is $\mathcal{O}(\log D)$ where D is the Hilbert space dimension. So the overall execution time scaling of this method is $\mathcal{O}(MD \log D)$.

The system used to benchmark both methods is composed of 8 sites. Here the Tesla rack was used to execute the benchmark, so the behaviour of the Bisection Method can be examined at higher particle numbers, as the consumer graphics card suffers from kernel timeouts at such amounts of particles.

The durations of the matrix-vector products can be seen in figure 5.8. The fitting function is due to the considerations above given by

$$t(N) = bD(N,M)\log(D(N,M)) = b\frac{\Gamma(N+M)}{\Gamma(N+1)\Gamma(M)}\log\left(\frac{\Gamma(N+M)}{\Gamma(N+1)\Gamma(M)}\right). \tag{5.5}$$

The duration of the Bisection Method is divided by the one of the presented algorithms, which is called 'speedup'. This quantity should scale with $\log D$. However, this is obviously not the case as can be seen in figure 5.9. Presumably the search algorithm behaves non-deterministically in relation to the total particle number N, as it heavily relies on branching, which results in unpredictable dependencies to the outcome of other searches carried out by other threads of a warp. Also the random accesses to the array containing all hash values are uncached. This is also the reason for the considerable performance difference even at small N.

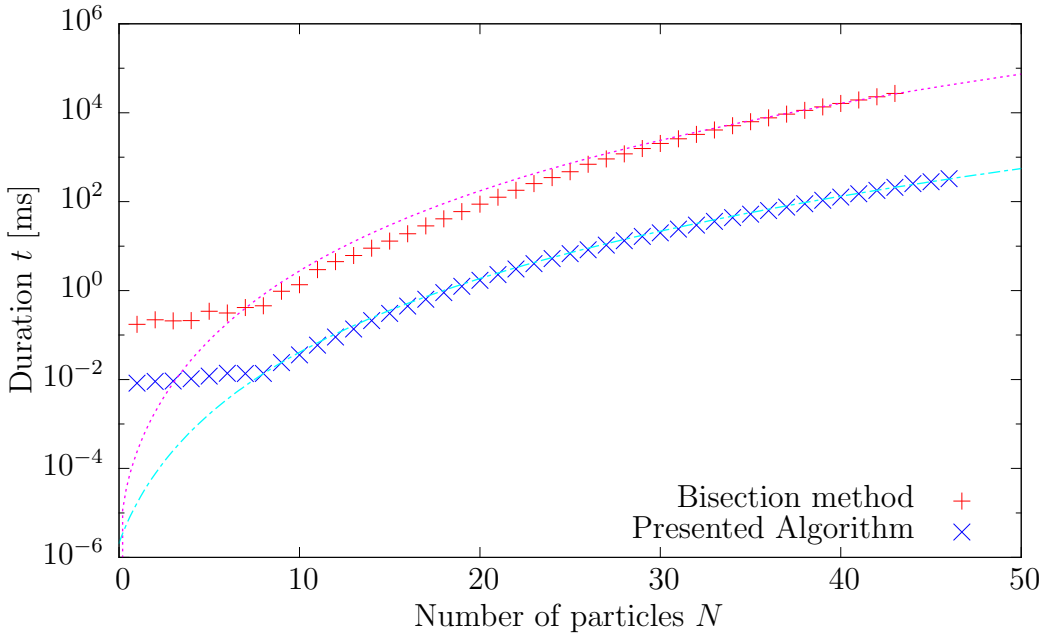


Figure 5.8.: Performance differences between the presented and the Bisection Method for a system with 8 sites. The graphics card used for the measurements is the Tesla C2070.

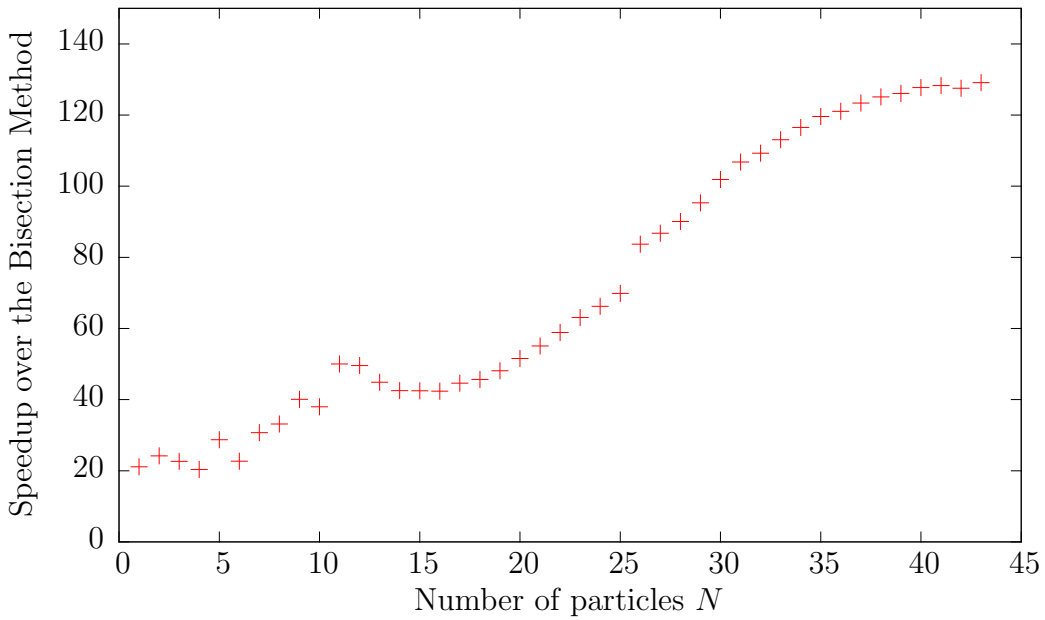


Figure 5.9.: The speedup between the Bisection Method and the presented algorithm for $M = 8$.

6. Applications

To proof the functionality of the presented algorithm, results of simulations are compared to an analytical solution and the results of the Bogoliubov Backreaction method [16, 17].

6.1. Double well system with one particle

The first system used for comparison consists of two potential wells of the same depth and a single particle. The potentials are constant. Due to the simplicity of this system an analytical solution for the state evolution exists.

The two on-site energies can be dropped, because both have the same value, so they only add a zero-point energy to the Hamiltonian. The interaction term can also be set to zero, as there is no two body interaction for just one particle.

$$H = -J\left(b_1^\dagger b_2 + b_2^\dagger b_1\right). \tag{6.1}$$

It is easy to find the two eigenstates

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}\left(|1,0\rangle + |0,1\rangle\right), \tag{6.2a}$$

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}\left(|1,0\rangle - |0,1\rangle\right) \tag{6.2b}$$

with eigenvalues $-J$ and J respectively. The Schrödinger equation (4.1) can thus be solved by the ansatz

$$|\psi\rangle = A\mathrm{e}^{iJt}\,|\psi_1\rangle + B\mathrm{e}^{-iJt}\,|\psi_2\rangle\,, \tag{6.3}$$

where $\hbar$ is set to 1. The expectation values for the operators n_0 and n_1 are

$$\langle n_0\rangle = \cos^2(Jt), \tag{6.4a}$$

$$\langle n_1\rangle = \sin^2(Jt) \tag{6.4b}$$

with the initial state set to $|\psi(0)\rangle = |1,0\rangle$. These expectation values are compared to the simulated ones in figure 6.1 with $J = 0.4$. It can be clearly seen that the resulting data points provided by the algorithm closely follow the analytical solution.

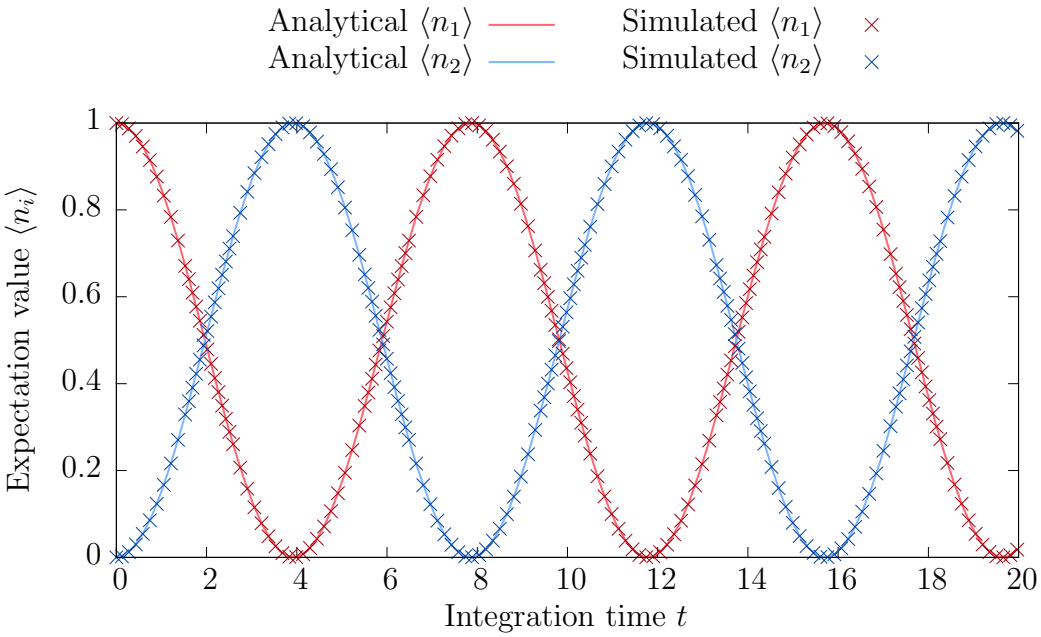


Figure 6.1.: Comparison of an analytical expression of the expectation values $\langle n_i \rangle$ to simulated ones. The used system consists out of two sites and one particle. The initial state is $|1, 0\rangle$. The approximated accumulation error at the end of the integration is $9.97 \cdot 10^{-11}$.

To verify the correctness of the presented method with time-dependent potential, a dynamic tunnel rate J is introduced into the Hamiltonian (6.1). For

$$J(t) = \mathrm{e}^{-at} \tag{6.5}$$

there exists a simple analytical time evolution of the state

$$|\psi\rangle = C \exp\left(-\frac{i}{a}\mathrm{e}^{-at}\right)|\psi_1\rangle + D \exp\left(\frac{i}{a}\mathrm{e}^{-at}\right)|\psi_2\rangle \tag{6.6}$$

with $\hbar = 1$. The physical interpretation of the Hamiltonian is a steady separation of the potential wells, as the overlap J of the two basis wave functions lessens with time. With the initial condition $|\psi(0)\rangle = |1, 0\rangle$, the expectation values of the occupation numbers become

$$\langle n_0 \rangle = \cos^2\left(\frac{\mathrm{e}^{-at} - 1}{a}\right), \tag{6.7a}$$

$$\langle n_1 \rangle = \sin^2\left(\frac{\mathrm{e}^{-at} - 1}{a}\right). \tag{6.7b}$$

For $a = 0.3$ the Eqs. (6.7a) and (6.7b) are plottet along with the simulated points given by the presented algorithm in figure 6.2. It is apparent that the algorithm presented in chapter 4 yields the correct time evolution of the initial state.

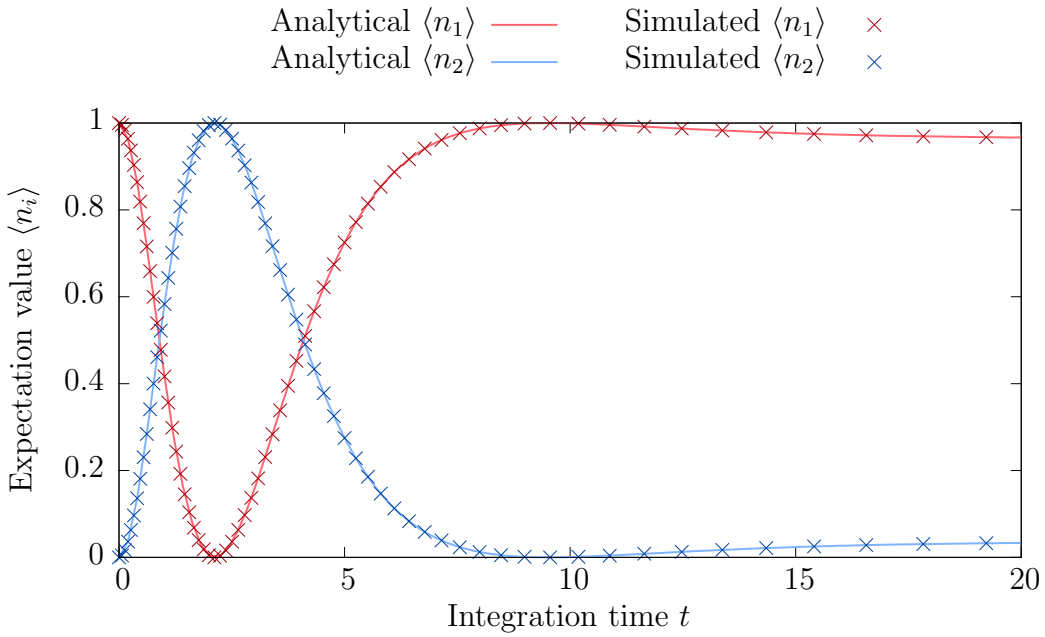


Figure 6.2.: Analytical and simulated expectation values $\langle n_i \rangle$ for a system consiting of two sites and one particle with a non-constant tunnel rate $J = \mathrm{e}^{-0.3t}$. The expectation values $\langle n_i \rangle$ tend to some constant, as the state freezes for $J \to 0$.

6.2. Comparison to the Bogoliubov-Backreaction method

In this section the time evolution of the particle number expectation values $\langle n_i \rangle$ provided by the algorithm described in chapter 4 is compared to the one of the Bogoliubov Backreaction (BBR) method [16, 17]. The simulated system consist of 4 sites and static potentials. During the integration $\hbar = 1$ applies. The initial state is prepared as an in phase mean-field state [17]

$$|\psi\rangle = \frac{1}{\sqrt{N!}} \sum_{n_1+n_2+\ldots+n_M=N} \binom{N}{n_1, n_2, \ldots, n_M} \prod_{m=1}^{M} \left(C_m b_m^\dagger\right)^{n_m} |0\rangle \tag{6.8}$$

with C_m^2 being the initial normalized particle densities $\langle n_m(t=0) \rangle / N$, which are set to

$$C_1^2 = 130/190, \tag{6.9a}$$
$$C_2^2 = 7/190, \tag{6.9b}$$
$$C_3^2 = 3/190, \tag{6.9c}$$
$$C_4^2 = 50/190. \tag{6.9d}$$

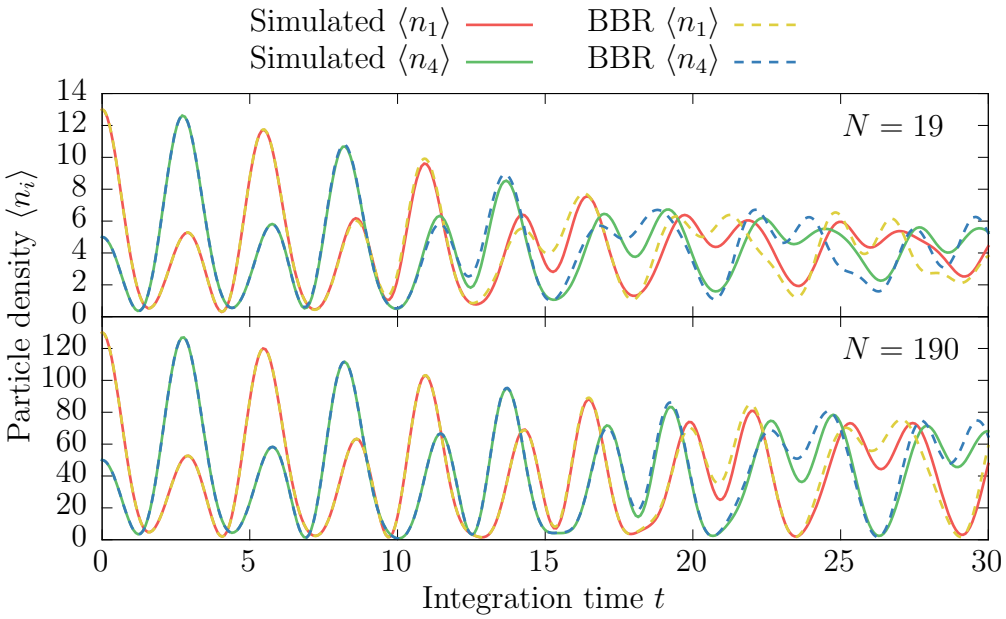


Figure 6.3.: Outer particle densities $\langle n_1 \rangle$ and $\langle n_4 \rangle$ over time provided by the BBR method and the presented algorithm for $N = 19$ and $N = 190$. The nonlinearity strength g is set to 1.

The system parameters read

$$J_{k,k+1} = 1, \tag{6.10a}$$
$$V_k = 0, \tag{6.10b}$$
$$U_k = g/(N-1), \tag{6.10c}$$

where g is the nonlinearity strength in the Gross-Pitaevskii equation [15]. Two different total particle numbers 19 and 190 are examined. The embedded Runge-Kutta method was used to ensure that the error during integration stays small. The results are shown in figures 6.3 and 6.4 with $g = 1$. The BBR method data is taken from [17]. There the BBR method is evaluated to second-order terms, which is sufficient to simulate the system accurately at small time scales [25]. At first particle density curves of the BBR method and the ones simulated with algorithm A.3 follow each other closely. At around $t = 10$ to $t = 20$ they start to diverge. With larger particle numbers this deviation point occurs later than at lower ones, as the BBR method is at the lowest order for $N \to \infty$ a mean-field approximation. Therefore the BBR method backs the validity of the presented algorithm, as their behaviour for small integration times are matching.

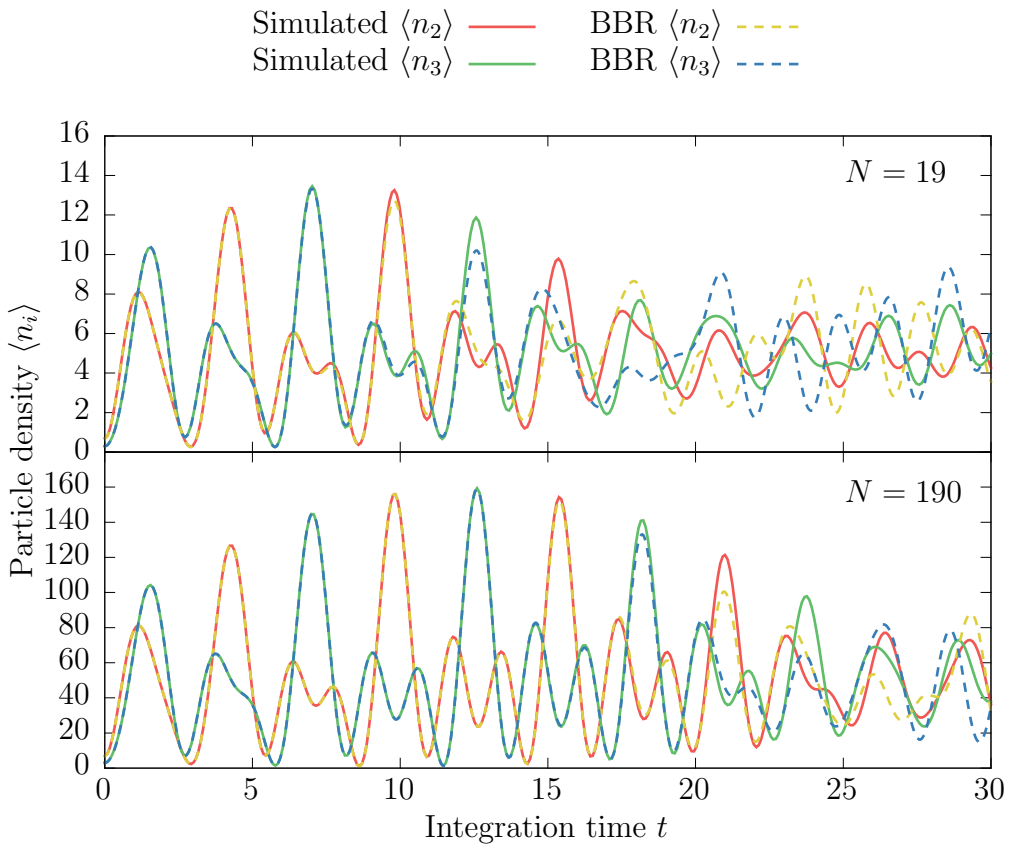


Figure 6.4.: Inner particle densities $\langle n_2 \rangle$ and $\langle n_3 \rangle$ over time provided by the BBR method and the presented algorithm for $N = 19$ and $N = 190$. The nonlinearity strength g is set to 1.

7. Summary and future work

7.1. Summary

The present thesis describes an efficient algorithm to determine the time evolution of the Bose-Hubbard model with time-dependent potentials. The Bose-Hubbard model was interpreted as a system of particles in a periodic structure at absolute zero. The system described here had M potential wells and N particles.

The main part of the algorithm presented here was the calculation of a matrix-vector product, which was carried out matrix-free, i.e. without storing the matrix explicitly. Due to the 'on-the-fly' nature of the matrix-vector product, it was straightforward to incorporate a time dependence of the potentials into the system. The order of Fock states was an important part of the derivation of this algorithm. Using combinatorics a state indexing routine was found. For neighbouring site transitions of particles between two states the access time to the amplitudes of these states is of order $\mathcal{O}(1)$. Therefore the time complexity of the matrix-vector product is $\mathcal{O}(MD)$ with the Hilbert space dimension D. It is also possible to use the neighbouring site transitions multiple times to perform an arbitrary transition, so more complicated operators can be implemented. The time evolution itself was carried out using two different Runge-Kutta methods, namely the classical 4th order Runge-Kutta method and an embedded Runge-Kutta method [22], which can provide 4th and 5th order time steps simultaneously.

A performance improvement was achieved by executing the presented algorithm in parallel utilizing a graphics card as a general purpose computational device. The GPU was chosen as a platform for these calculations, as it is capable to execute a large amount of tasks simultaneously. The implementation was done using CUDA, an API provided by Nvidia. The performance of this implementation was compared to cuSPARSE, a library for parallel sparse matrix computations also developed by Nvidia. The presented algorithm was found to outperform the well optimized cuSPARSE implementation by a factor of 165% on a consumer level graphics card.

To prove the relevancy of the GPU implementation, a CPU one of the very same algorithm was made. The execution time of the CPU implementation surpassed the GPU one by a factor of 200% with consumer level hardware and 2300% on a professional graphics card.

The presented algorithm was also compared to a different method of matrix-free matrix-

vector multiplication involving hash functions and a Bisection search to retrieve the index of a given Fock state [19]. Due to this required search algorithm, the time complexity of this method is $\mathcal{O}(MD \log D)$, which has a worse scaling behaviour than the presented algorithm.

Last but not least, the functionality of the presented method was demonstrated by application to different systems and comparison to another work, where the Bogoliubov-Backreaction method was implemented [17].

7.2. Future work

As the amount of particles and sites increases, the memory necessary to store the state of the system inflates exponentially. Often the GPU memory capacity is smaller than the one of the main memory. In that case it would be beneficial to divide the state vector into multiple pieces and stream those to GPU memory to perform parts of the matrix-vector multiplication similar to [26] and [27]. Afterwards the intermediate results are summed up to yield the final matrix-vector product. That way, only a user defined amount of device memory is spend, whereas the whole state vector is kept on the main memory. A problem with this solution could be the additional bandwidth constraint of the CPU-GPU communication, which is not present in the method presented in this thesis, because the state vector is kept on the device memory at all times.

One could follow a different track by applying the presented state enumeration method to other Hamiltonians, like such with non-constant particle numbers. This can be done by dividing the state vector into parts of constant particle numbers. Transitions between states with different particle numbers can then be computed by offsetting the indexing method used in this thesis by the dimensions of sub-Hilbert spaces with constant particle numbers.

A. Appendix

Listing A.1: Index generation algorithm.

```
int getIndex(int* particle, int len /*tuple array and length*/,
    int N, int M)
{
    int remaining_particles = N;
    int remaining_sites = M;
    int result = D(N, M);
    for(int i = 0; i < len; ++i)
    {
        int action_i = particle[i];
        if(action_i == 0)
        {
            //if the value of the tuple at i is 0,
            //then skip over all tuples with value 1 at i
            //and the same beginning sequence of 1 and 0
            //as the current tuple
            //this is done by subracting the dimension of
            //the set of subtuples, consisting of the
            //remaining number of particles and sites
            result -= D(remaining_particles - 1,
                remaining_sites);
            //and move to the next site
            remaining_sites--;
        }
        else
        {
            //if the current action is 1,
            //then remove a particle
            remaining_particles--;
        }
    }
    return result;
}
```

Listing A.2: Algorithm for determining the index difference, when a particle transitions to a neighbouring site.

```
int getDelta(int* particle, int len, int p /*exchange position*/,
    int N, int M)
{
    int remaining_particles = N;
    int remaining_sites = M;
    //calculate remaining particles and sites up to position p
    for(int j = 0; j < p - 1; ++j)
    {
        int action_i = particle[j];
        if(action_i == 0)
            remaining_sites--;
        else
            remaining_particles--;
    }
    //calculate the index difference
    return D(remaining_particles - 1, remaining_sites - 1);
}
```

Listing A.3: The complete matrix-multiplication algorithm.

```
cmplx HamiltonianMultiplication(int i /*index of the output
    amplitude*/,
        cmplx* state /*state vector*/,
        int* occupation_numbers_1 /*2D-array containing all
           occuation number tuples*/,
        int N, int M, int* Delta_1 /*Delta-function as a 2D-look-
           up table*/,
        double* SQRT_1 /*2D look-up table, that stores all
           possible sqrt((n_k+1)n_j) coefficients for j=k+1*/,
        double* V /*array containing all on-site energies*/,
        double* U /*all interaction potentials*/,
        double* J /*all hopping amplitudes*/
)
{
  cmplx result = cmplx(0.0, 0.0); //initialize the complex return
      value

  //initialize all needed variables
  double u_accumulation = 0.0;
  double v_accumulation = 0.0;
  int remaining_particles = N;
  int n_now = getOccupationNumberAtSite(0, i,
     occupation_numbers_1, M);
  int n_next;
  //iterate over hopping terms, on-site and interaction energies
  for(int m = 0; m < M - 1; ++m)
  {
    int remaining_sites = M - m;
    remaining_particles -= n_now;
    n_next = getOccupationNumberAtSite(m+1, i,
       occupation_numbers_1, M);

    //calculate hopping terms
    if(n_now != 0)
    {
      //forward exchange (...,1,0,...) -> (...,0,1,...)
      int delta = getDelta(remaining_particles, remaining_sites,
         Delta_1, M);
      //calculate sqrt((n_next + 1)*n_now)
      double sqrt_coeff = getSQRT((n_next + 1)*n_now, SQRT_1);
      //delta gets subtracted, because the dimensions
      //get also subtracted for the computation of the indices
      result += -J[m] * sqrt_coeff * state[i-delta];
```

```
36      }
37      if(n_next != 0)
38      {
39        //backward exchange (...,0,1,...) -> (...,1,0,...)
40        int delta = getDelta(remaining_particles-1, remaining_sites
              , Delta_l, M);
41        //the -1 comes from the fact, that the 0 precedes the 1
              when a backward exchange is calculated
42        //calculate sqrt((n_now + 1)*n_next)
43        double sqrt_coeff = getSQRT((n_now + 1)*n_next, SQRT_l);
44        result += -J[m] * sqrt_coeff * state[i+delta];
45      }
46
47      //accumulate on-site energies
48      v_accumulation += V[m] * n_now;
49
50      //accumulate interaction energies
51      u_accumulation += U[m] * n_now * (n_now - 1);
52
53      n_now = n_next;
54    }
55
56    //add last terms
57    v_accumulation += V[M-1] * n_now;
58    u_accumulation += U[M-1] * n_now * (n_now - 1);
59
60    // return result of the vector multiplication for the i'th row
61    return result + (0.5 * u_accumulation + v_accumulation) * state
          [i];
62  }
```

Listing A.4: The matrix-vector multiplication kernel.

```
//N and M are here defined globally at compilation time, to use static shared memory
#define N 190
#define M 4

__shared__ double U_shared[M];
__shared__ double V_shared[M];
__shared__ double J_shared[M];

__global__ void HamiltonianMultiplication_kernel(cmplx* state /*input state vector*/,
    cmplx* result /*output vector*/,
    int dim /*dimension of the hilbert space*/,
    int num_sequential /*variable, which controls the number of output vector rows a thread computes*/,
    unsigned int* occupation_numbers_l, double* SQRT_l,
    int* Delta_l, double* U, double* V, double* J)
{
    //initialize indexing variables
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int bDim = blockDim.x;
    int gDim = gridDim.x * bDim;

    //initialize needed arrays

    //SEPARATION is a global variable, which is dependent on the memory usage per occupation number
    //Example: M=4 with 8 bit per occupation number leads to a SEPERATION value of 1, as an unsinged int uses 32 bits of space
    unsigned int used_occupation_numbers[SEPARATION];
    if(tid < M)
    {
      U_shared[tid] = U[tid];
      V_shared[tid] = V[tid];
    }
    if(tid < M - 1)
      J_shared[tid] = J[tid];

    //synchronize threads inside a warp, to make sure, that all shared variables are loaded
    __syncthreads();
```

```
37
38      for (int g = 0; g < num_sequential; ++g)
39      {
40          //iterate over a range of output rows
41          int i = tid + bid * bDim + g * gDim;
42
43          //load all necessary occupation numbers
44          for (int j = 0; j < SEPARATION; ++j)
45              used_occupation_numbers[j] = occupation_numbers_l[i *
                    SEPARATION + j];
46
47          //only process the i'th row if it lies inside the bounds
                of the state vector
48          if (i < dim)
49          {
50              //compute the dot product of the state vector with
                    the i'th row of the hamiltonian and write the
                    result to the output vector
51              result[i] = HamiltonianMultiplication_GPU(i, state,
                    used_occupation_numbers, Delta_l, SQRT_l);
52          }
53      }
54  }
```

Listing A.5: The 4th order Runge-Kutta kernel and host code for execution.

```
__global__ void RungeKutta4_kernel(cmplx* A, cmplx* B /*these vectors get switched to save memory*/, double h /*stepsize*/, int stage /*indicates at which stage the Runge-Kutta computation is*/,
                   cmplx* state, cmplx* result, int dim, int num_sequential, unsigned int* occupation_numbers_l, double* SQRT_l, int* Delta_l, double* U, double* V, double* J)
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int bDim = blockDim.x;
    int gDim = gridDim.x * bDim;unsigned int used_occupation_numbers[SEPARATION];
    if(tid < M)
    {
      U_shared[tid] = U[tid];
      V_shared[tid] = V[tid];
    }
    if(tid < M - 1)
      J_shared[tid] = J[tid];

    __syncthreads();

    for (int g = 0; g < num_sequential; ++g)
    {
        int i = tid + bid * bDim + g * gDim;

        for (int j = 0; j < SEPARATION; ++j)
            used_occupation_numbers[j] = occupation_numbers_l[i * SEPARATION + j];

        if (i < dim)
        {
            if(stage == 0)
            {
                //compute the first Runge-Kutta vector k_1=-i*H(t_0)*v
                cmplx k1 = cuCmplx(0.0, -1.0) * HamiltonianMultiplication_GPU(i, state, used_occupation_numbers, Delta_l, SQRT_l);
                //calculate the next evaluation position for the next k variable
```

```
32                  A[i] = state[i] + h* 0.5 * k1;
33                  //accumulate the result
34                  result[i] = state[i] + ((h * k1) / 6.0);
35              }
36              if(stage == 1)
37              {
38                  cmplx k2 = cuCmplx(0.0, -1.0) *
                       HamiltonianMultiplication_GPU(i, A,
                       used_occupation_numbers, Delta_l, SQRT_l);
39                  B[i] = state[i] + h * 0.5 * k2;
40                  result[i] = result[i] + ((h * k2) / 3.0);
41              }
42              if(stage == 2)
43              {
44                  cmplx k3 = cuCmplx(0.0, -1.0) *
                       HamiltonianMultiplication_GPU(i, B,
                       used_occupation_numbers, Delta_l, SQRT_l);
45                  A[i] = state[i] + h * k3;
46                  result[i] = result[i] + ((h * k3) / 3.0);
47              }
48              if(stage == 3)
49              {
50                  cmplx k4 = cuCmplx(0.0, -1.0) *
                       HamiltonianMultiplication_GPU(i, A,
                       used_occupation_numbers, Delta_l, SQRT_l);
51                  //copy the result back to the state vector, so
                       the next Runge-Kutta step can be computed in
                       the same way
52                  state[i] = result[i] + ((h * k4) / 6.0);
53              }
54          }
55      }
56  }
57
58  ...
59  typedef double(*func)(double, int);
60  ...
61
62  void RungeKutta::calculateRungeKuttaStep(double& t /*current
        integration time*/, double h, int dim, int num_sequential, int
        grid_size, func U_function, func V_function, func J_function)
63  {
64      //all variables needed are stored inside the RungeKutta class
65
```

```
66      //calculate the U, V and J arrays by evaluating the given
           functions and cudaMemcpy them onto the device memory
67      //the functions U_function, V_function and J_function require
            the integration time t and an integer, representing the
           site of the output potential value
68      calculate_UVJ(t, U_function, V_function, J_function);
69      //execute the first Runge Kutta stage
70      //BLOCKSIZE is a variable defined at compilation time
71      RungeKutta4_kernel<<<grid_size, BLOCKSIZE >>>(A, B, h, 0,
           state, result, dim, num_sequential, occupation_numbers_l,
           SQRT_l, Delta_l, U, V, J);
72      calculate_UVJ(t + h * 0.5, U_function, V_function, J_function
           );
73      //compute the next two stages
74      RungeKutta4_kernel<<<grid_size, BLOCKSIZE >>>(A, B, h, 1,
           state, result, dim, num_sequential, occupation_numbers_l,
           SQRT_l, Delta_l, U, V, J);
75      RungeKutta4_kernel<<<grid_size, BLOCKSIZE >>>(A, B, h, 2,
           state, result, dim, num_sequential, occupation_numbers_l,
           SQRT_l, Delta_l, U, V, J);
76      //add a time step
77      t += h;
78      calculate_UVJ(t, U_function, V_function, J_function);
79      RungeKutta4_kernel<<<grid_size, BLOCKSIZE >>>(A, B, h, 3,
           state, result, dim, num_sequential, occupation_numbers_l,
           SQRT_l, Delta_l, U, V, J);
80  }
```

Bibliography

[1] Chris A Mack. "Fifty years of Moore's law". In: *IEEE Transactions on semiconductor manufacturing* 24.2 (2011), pp. 202–207.

[2] Pierre Astier and Reynald Pain. "Observational evidence of the accelerated expansion of the universe". In: *Comptes Rendus Physique* 13.6 (2012), pp. 521–538.

[3] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[4] Shekhar Borkar. "Design challenges of technology scaling". In: *IEEE micro* 19.4 (1999), pp. 23–29.

[5] Shuai Che et al. "A performance study of general-purpose applications on graphics processors using CUDA". In: *Journal of parallel and distributed computing* 68.10 (2008), pp. 1370–1380.

[6] Karl Rupp. *CPU, GPU and MIC Hardware Characteristics over Time*. June 21, 2013. URL: `https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/` (visited on 07/05/2016).

[7] Dieter Jaksch et al. "Cold bosonic atoms in optical lattices". In: *Physical Review Letters* 81.15 (1998), p. 3108.

[8] Hagen Kleinert. "Field Formulation of Many-Body Quantum Physics". In: *Particles and quantum fields*. World Scientific Pub, 2016, pp. 82–102. ISBN: 978-981-4740-89-0.

[9] Joseph Callaway. *Quantum theory of the solid state*. Academic Press, 2013.

[10] Nicola Marzari, Ivo Souza, and David Vanderbilt. "An introduction to maximally-localized Wannier functions". In: *Psi-K Newsletter* 57 (2003), p. 129.

[11] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Sept. 1, 2015. URL: `docs.nvidia.com/cuda/cuda-c-programming-guide/` (visited on 08/05/2016).

[12] Kamran Karimi, Neil G. Dickson, and Firas Hamze. "A Performance Comparison of CUDA and OpenCL". In: *CoRR* abs/1005.2581 (2010). URL: `http://arxiv.org/abs/1005.2581`.

[13] Nam Le. *NVidia CUDA : my experience with High Performance Computing*. URL: `http://lenam701.blogspot.de/2012/02/nvidia-cuda-my-experience-with-high.html` (visited on 08/05/2016).

[14] NVIDIA Corporation. *cuSPARSE library*. Sept. 1, 2015. URL: `http://docs.nvidia.com/cuda/cusparse/` (visited on 08/05/2016).

[15] Franco Dalfovo et al. "Theory of Bose-Einstein condensation in trapped gases". In: *Reviews of Modern Physics* 71.3 (1999), p. 463.

[16] I. Tikhonenkov, J. R. Anglin, and A. Vardi. "Quantum dynamics of Bose-Hubbard Hamiltonians beyond the Hartree-Fock-Bogoliubov approximation: The Bogoliubov back-reaction approximation". In: *Phys. Rev. A* 75 (1 Jan. 2007), p. 013613. URL: http://link.aps.org/doi/10.1103/PhysRevA.75.013613.

[17] Johannes Reiff. *Beschreibung $\mathcal{PT}$-symmetrischer Bose-Einstein-Kondensate mit einem Vier-Mulden-Potential und der Bogoliubov-Backreaction-Methode*. German. 1. Institut für Theoretische Physik der Universität Stuttgart, Aug. 15, 2016.

[18] Jane K Cullum and Ralph A Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations: Vol. 1: Theory*. Vol. 41. Siam, 2002.

[19] JM Zhang and RX Dong. "Exact diagonalization: the Bose–Hubbard model as an example". In: *European Journal of Physics* 31.3 (2010), p. 591.

[20] Peter J Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge University Press, 1994.

[21] J. C. Butcher. "A History of Runge-Kutta Methods". In: *Appl. Numer. Math.* 20.3 (Mar. 1996), pp. 247–260. ISSN: 0168-9274. URL: http://dx.doi.org/10.1016/0168-9274(95)00108-5.

[22] William H Press et al. "Adaptive Stepsize Control for Runge-Kutta". In: *Numerical recipes in FORTRAN: the art of scientific computing*. Vol. 2. Cambridge University Press Cambridge, 1992, pp. 714–722. ISBN: 0-521-43108-5.

[23] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU gems* 3.39 (2007), pp. 851–876.

[24] *std::chrono::steady_clock::now*. May 31, 2015. URL: http://en.cppreference.com/w/cpp/chrono/steady_clock/now (visited on 08/09/2016).

[25] Dennis Dast et al. "Quantum master equation with balanced gain and loss". In: *Physical Review A* 90.5 (2014), p. 052120.

[26] Jing Wu and Joseph Jaja. "Achieving Native GPU Performance for Out-of-Card Large Dense Matrix Multiplication". In: *Parallel Processing Letters* 26.02 (2016), p. 1650007.

[27] LM Itu et al. "GPU Enhanced Stream-Based Matrix Multiplication". In: *Bulletin of the Transilvania University of Brasov, Series I: Engineering Sciences* 5.2 (2012).

Zusammenfassung in deutscher Sprache

1. Zusammenfassung

Die vorliegende Arbeit beschreibt einen effizienten Algorithmus zur Berechnung der Zeitentwicklung des Bose-Hubbard Modells mit zeitabhängigen Potentialen. Das Bose-Hubbard Modell wurde interpretiert als ein System von Teilchen in einer periodischen Gitterstruktur bestehend aus M Potentialtöpfen und N Teilchen am absoluten Temperaturnullpunkt.

Hauptbestandteil des vorgestellten Algorithmus ist eine Matrix-Vektor-Multiplikation, welche „Matrix-frei“ ausgeführt wurde, d.h. ohne die Matrix explizit zu speichern. Wegen der dynamischen Natur dieses Matrix-Vektor-Produkts war es unkompliziert eine Zeitabhängigkeit der Potentiale in das System einzubauen. Die Reihenfolge der Fock-Zustände war ein wichtiger Teil der Herleitung des Algorithmus. Mithilfe von Kombinatorik wurde eine Zustandsnummerierungsmethode gefunden. Für Nächste-Nachbar-Übergänge von Teilchen zwischen zwei Zuständen ist die Zugriffszeit auf die Amplituden dieser Zustände von der Ordnung $\mathcal{O}(1)$. Deshalb ist die Zeitkomplexität des Matrix-Vektor-Produkts von der Ordnung $\mathcal{O}(MD)$, wobei D die Dimension des Hilbertraums ist. Es ist außerdem möglich durch hintereinander Ausführung von mehreren Nächste-Nachbar-Sprüngen beliebige Übergänge zu berechnen, sodass auch kompliziertere Operatoren damit umgesetzt werden können. Die Zeitentwicklung selbst wurde mithilfe von zwei verschiedenen Runge-Kutta-Verfahren durchgeführt, der klassischen Runge-Kutta-Methode 4ter Ordnung und einer eingebetteten Runge-Kutta-Methode [22]. Die eingebettete Runge-Kutta-Methode bestimmt Zeitschritte 4ter und 5ter Ordnung gleichzeitig.

Durch parallele Ausführung des vorgestellten Algorithmus auf einer Grafikkarte wurde eine Verbesserung der Laufzeit erzielt. Die GPU wurde als Plattform für diese Berechnungen gewählt, da sie in der Lage ist eine große Anzahl an Rechnungen gleichzeitig auszuführen. Die Implementation wurde mittels CUDA realisiert, einer von Nvidia bereitgestellten API. Die Performanz dieser Implementation wurde mit cuSPARSE verglichen, einer Bibliothek für dünn besetzte Matrizen, welche ebenfalls von Nividia entwickelt wurde. Der vorgestellte Algorithmus übertraf die Leistung der wohl optimierten cuSPARSE Implementation um einem Faktor von 165% auf einer kommerziell erhältlichen Grafikkarte.

Um die Relevanz der GPU Implementation zu zeigen, wurde eine CPU Umsetzung desselben Algorithmus erstellt. Die Laufzeit der CPU Implementation übertraf die der GPU um einen Faktor von 200% auf einer gewöhnlichen und um 2300% auf einer professionellen Grafikkarte.

Der vorgestellte Algorithmus wurde außerdem mit einer anderen Matrix-freien Methode zur Matrix-Vektor-Multiplikation verglichen, welche zur Berechnung des Index eines Fock-Zustands eine Hashfunktion und eine Bisektion erfordert [19]. Wegen des benötigten Suchalgorithmuses ist die Zeitkomplexität von der Ordnung $\mathcal{O}(MD \ln D)$. Somit hat diese Methode ein schlechteres Skalierungsverhalten als der vorgestellte Algorithmus.

Zum Schluss wurde die Funktionalität der beschriebenen Methode demonstriert, indem sie auf unterschiedliche Systeme angewandt und außerdem mit Ergebnissen einer anderen Arbeit verglichen wurde, in der die Bogoliubov-Backreaction-Methode verwendet wurde [17].

2. Ausblick

Der Speicherverbrauch des Zustandsvektors vergrößert sich bei einer Zunahme der Anzahl der Teilchen oder Potentialtöpfe exponentiell. Meist ist aber die Speicherkapazität der GPU kleiner als die des Arbeitsspeichers. In diesem Fall ist es vorteilhaft den Zustandsvektor aufzuteilen und diese zur GPU zu „streamen“, sodass immer nur ein Teil der Matrix-Vektor-Multiplikation ausgeführt wird, ähnlich zu [26] und [27]. Danach werden diese Zwischenergebnisse aufsummiert um das endgültige Matrix-Vektor-Produkt zu erhalten. Somit wird nur eine benutzerdefinierte Menge an Grafikkartenspeicher verbraucht, während der gesamte Zustandsvektor auf dem Arbeitsspeicher liegt. Ein Problem dieser Lösung ist vermutlich der zusätzliche Bandbreitenbedarf der CPU-GPU Kommunikation, welcher in dem vorgestellten Algorithmus nicht auftritt, da der Zustandsvektor stets auf dem Grafikkartenspeicher gehalten wird.

Möglich wäre auch eine Anwendung der vorgestellten Zustandsnummerierung auf andere Hamilton-Operatoren, z. B. mit variablen Gesamtteilchenzahlen. Dies kann durch Aufteilen in Blöcke mit konstanter Teilchenzahl erreicht werden. Übergänge zwischen Zuständen mit unterschiedlichen Teilchenzahlen können ebenfalls mit der in dieser Arbeit entwickelten Zustandsnummerierung unter Aufsummierung der Dimensionen von Hilbert-Unterräumen realisiert werden.

Danksagung

An dieser Stelle will ich mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben.

Als Erstes will ich mich besonders bei meinem Prüfer Prof. Dr. Jörg Main bedanken, der sich immerzu Zeit für mich nahm und diese Arbeit erst möglich machte. Auch danke ich meinem Betreuer Daniel Dizdarevic für die rege Zusammenarbeit und für seinen hilfreichen Rat, vor allem in der Schreibphase. Obendrein danke ich meinen Zimmerkollegen Christoph Lohrmann und Johannes Reiff für die angenehme Arbeitsatmosphäre. Dem Letzteren will ich besonders für sein endloses Wissen über LaTeX und C++ danken, welches er gerne mit mir geteilt hat.

Mein Dank gilt auch meinen Kommilitonen und Freundeskreis, bestehend aus Annette Böhme, Simon Klein, Pascal Strobel, Tim Strobel und Mario Zinßer, für die amüsante Gesellschaft und gelegentlichen durchaus produktiven Lernstunden.

Zum Schluss danke ich meiner Familie, die mich von jeher stets unterstützt hat.

Erklärung

Ich versichere,

- dass ich diese Bachelorarbeit selbständig verfasst habe,
- dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe,
- dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist,
- und dass das elektronische Exemplar mit den anderen Exemplaren übereinstimmt.

Stuttgart, den 15. August 2016 *Kirill Alpin*