Università degli studi di Padova

Dipartimento di Fisica e Astronomia " Galileo Galilei "

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Fisica

Large-scale Classical Simulation of Quantum Systems Using the Trotter–Suzuki Decomposition

Relatore esterno: Antonio Acín
Correlatore esterno: Peter Wittek
Relatore interno: Giuseppe Vallone

Laureando
Luca Calderaro

Anno Accademico 2014/2015

Dedicated to Sara.

Abstract

Many theoretical studies and experimental results rely on the use of numerical analysis for the solution of the Schrödinger equation. Indeed, for nontrivial quantum systems, a complete solution of the dynamics is difficult to achieve analytically. We extended the implementation of a highly optimized solver to simulate the evolution of a wave function on a 2D lattice. We also implemented the imaginary time evolution to approximate the ground state. The dynamics of the system is now described by a Hamiltonian that includes an external potential and a contact interaction term. The algorithm is based on the second-order Trotter–Suzuki approximation and it is implemented on CPU and GPU kernels that run efficiently on a cluster. We proved the accuracy of the code solving the Gross–Pitaevskii equation for a Bose–Einstein condensate and reproducing the experimental results, obtained at NIST, of the soliton dynamics in a cloud of sodium atoms. The code is available under an open source license, and it is exposed as an application program interface and a command–line interface. The code is also accessible in Python and MATLAB. Future development of the code include the extension to a 3D lattice, whereas the actual implementation can already find applications in ultracold atom physics.

Acknowledgement

This Thesis project has been carried out during my Erasmus internship at ICFO–The Institute of Photonic Sciences in Barcelona. Part of this work relied on the infrastructure provided by the Barcelona Supercomputing Center, sponsored by the Spanish Supercomputing Network (grant number: FY-2015-2-0023). Special thanks are given to my supervisors Peter Wittek and Toni Acín for having me in their research group and for all the great support they gave me. Also, I like to thank Pietro Massignan for his invaluable support in the physical application of the code. Thanks to all the ICFOnians who made my internship an enjoyable and productive experience.

Thanks to my parents and my sister for having raised me and always sustained me in my aspirations.

Contents

List of Figures

Chapter 1

Introduction

Since the creation of the first electronic digital computing device, physicists have used computers as a valuable tool for their research. The study and implementation of numerical methods to solve physical problems led to the growth of a new field of physics called computational physics. It is not surprising that this field of study has branches in every major field in physics: from computational mechanics to computational astrophysics, from computational condensed matter to computational particle physics. Mathematical models, developed to accurately describe natural phenomena, are often difficult to solve analytically. Typically, the construction of a physics model begins with the definition of the energy of the system, which contains the interactions between the components of the system and the kinetic energy of the particles – when these are allowed to move. This in turn leads to the action of the system and the equations of motion by means of the least action principle [18]. At this stage, depending on what system is under study, many features can already be found without considering the equations of motion. For instance, the phase transitions of a spin system in the canonical ensemble may be studied considering the partition function which in turn lets us calculate observables, like the magnetization as a function of the temperature and the external magnetic field [13]. These features regard the equilibrium properties of the system and may constitute a rather challenging analytical problem. Even more challenging is the study of the dynamical properties, in which one has to deal with the equations of motion. In particular, one would have to solve the Cauchy problem, in which the initial state of the system is given along with the equations.

The correctness of a physics model is evaluated comparing its results with the outcomes of the experiment. As we said, it is not always pos-

sible to get the results we need from the model by mean of an analytical method. A possible approach to tackle these problems is to use algorithms to numerically solve the equations.

Complicated systems lead to numerically intense simulations that require a great amount of computational resources. For this reason, the development of efficient code, which is able to take advantage of the computational resources available nowadays, is of fundamental importance. The lack of efficiency leads to long execution time that can extend to months or even years, making the simulations impracticable.

The most powerful computational facilities at our disposal are supercomputers. These machines consist of many single processing units connected with each other to share data. An algorithm can get the most out of these machines when it is able to use many processing units at the same time, parallelizing the tasks to be performed. Processing units can be distinguished into two main categories: central processing units (CPUs) and graphics processing units (GPUs). The former type dedicates most of their transistor count to improve sequential code performance, while the latter type takes a different approach, housing hundreds of simple execution units which run parallel code. Due to their advantageous features, GPUs are gaining popularity in the computational physics field. They are designed to perform simple calculations on large amount of data in parallel. This can lead to a great reduction of the execution time with respect to a sequential or parallel implementation on a CPU.

In this work we developed a solver for the Schrödinger equation that scales to massively parallel computing clusters. Our point of departure was the recent work of Wittek and Cucchietti [43]. In their work, they extended the single-node parallel kernels in Ref. [1] to use distributed resources. These kernels are cache optimized kernels for both CPUs and GPUs based on the second order Trotter–Suzuki decomposition [35], and implement a solver for the Schrödinger equation of a free particle.

We extended the code implementing the following features:

- The Hamiltonian includes the stationary external potential. The implementation is also able to solve the nonlinear Schrödinger equation, in which the nonlinear term is given by the delta-function interactions between bosonic particles – this is currently only implemented in the CPU kernel.

- Imaginary time evolution to approximate the ground state.

- Command-line interface and application programming interface for

flexible use.

- Python and MATLAB wrappers are provided.
- Unit testing framework was implemented.

The new version of the program has been already published and appears in a short paper: [Wittek, P. and Calderaro, L., Extended computational kernels in a massively parallel implementation of the Trotter–Suzuki approximation, Computer Physics Communications, August 2015.]. **The paper is appended at the end of this thesis**.

As an application of the extended implementation, we have been able to simulate the evolution of an interacting Bose–Einstein condensate described by the Gross–Pitaevskii equation. The simulation reproduced the experimental results in Ref. [7].

The content of the Thesis is organized in the following way. The second chapter introduces to the Trotter–Suzuki approximation. In the third we explicitly calculate the evolution operator that is implemented in our code. The fourth chapter gives the details of the algorithms used in our code, describing the optimization techniques. The fifth chapter presents the application to the interacting BEC and we compare our results with the experimental study of Ref. [7]. We conclude summarizing our achievements and outlining the future directions of research.

Chapter 2

Trotter–Suzuki Decomposition

Since the publication of the Trotter product formula [41], a great effort has been carried out by mathematicians to study possible approximations of the exponential operator. In particular, Masuo Suzuki has studied the higher-order approximation throughout his carrier, leading to major results on this subject [33, 34, 35, 36, 37, 38, 39].

The Trotter product formula for the exponential of two not necessarily commuting linear operators reads as follows:

$$\exp\left(A+B\right) = \lim_{n\to\infty}\left(\exp\left(\frac{A}{n}\right)\exp\left(\frac{B}{n}\right)\right)^{n}. \tag{2.1}$$

The Trotter–Kato theorem defines the properties that the operators A and B must satisfy for the Eq. (2.1) to hold [16]. In the simplest case, A and B can be seen as arbitrary $n\times n$ real or complex matrices, and Eq. (2.1) reduces to the Lie product formula [31]. The exponential of a generic operator is usually difficult to calculate, but whenever this operator can be expressed as a sum of two operators A and B, with easy to calculate exponentials, Eq. (2.1) provides a method to estimate $\exp\left(A+B\right)$. However, for practical purposes, this formula is not appropriate, since it requires to take the limit in n to infinity. On a practical side, we could calculate the right-hand side of the equation only for a finite value of n, leading to an approximation of the original problem. At this point, it becomes important to study what can be an efficient approximation of the exponential, and how to estimate the error.

2.1 Exponential Operators in Physics

First of all, let us discuss as to why we have to treat the exponential operator and why we need an approximation to deal with it. The exponential operator appears in various fields of physics as a formal solution of the differential equation of the following form:

$$\frac{\partial}{\partial t}x(t) = Mx(t), \tag{2.2}$$

where x is a function or a vector and M is a finite or infinite dimensional operator. Typical examples include the Schrödinger equation

$$\imath\hbar\frac{\partial}{\partial t}\psi(t) = H\psi(t), \tag{2.3}$$

the Hamiltonian equation

$$\frac{d}{dt}\begin{pmatrix}\vec{p}(t)\\ \vec{q}(t)\end{pmatrix} = H\begin{pmatrix}\vec{p}(t)\\ \vec{q}(t)\end{pmatrix}, \tag{2.4}$$

and the diffusion equation with a potential

$$\frac{d}{dt}P(x,t) = LP(x,t). \tag{2.5}$$

A formal solution of (2.2) is given in the form of the Green's function as

$$x(t) = G(t;0)x(0) = \exp(tM)\,x(0). \tag{2.6}$$

However, obtaining the Green's function $G(t;0) = \exp(tM)$ is as difficult as solving Eq. (2.2) in any other way. Another important instance of the exponential operator is the partition function in equilibrium quantum statistical physics:

$$Z = \mathrm{Tr}\left(\exp\left(-\beta H\right)\right). \tag{2.7}$$

The exponential operator, however, is hard to compute in most interesting cases. The computation of the exponential operator $\exp(xM)$ becomes straightforward when a basis that diagonalize the operator M is easy to obtain. In quantum many-body problems, however, the basis of the diagonalized representation is often nontrivial, because we are typically interested in the Hamiltonian with two terms or more that are mutually noncommutative. For example, the Ising model in a transverse field, written as follows:

$$H = -\sum_{\langle i,j\rangle} J_{ij}\sigma_i^z\sigma_j^z - \triangle\sum_i \sigma_i^x, \tag{2.8}$$

and the Hubbard model,

$$H = -t \sum_{\sigma=\uparrow,\downarrow} \sum_{\langle i,j \rangle} (c^\dagger_{i\sigma} c_{j\sigma} + c^\dagger_{j\sigma} c_{i\sigma}) + U \sum_i n_{i\uparrow} n_{i\downarrow}. \tag{2.9}$$

In the first example (2.8), the quantization axis of the first term is the spin z axis, while that of the second term is the spin x axis. The two terms are therefore mutually non-commutative. In the second example (2.9), the first term is diagonalizable in the momentum space, whereas the second term is diagonalizable in the coordinate space. In both examples, each term is easily diagonalizable. Since one quantization axis is different from the other, the diagonalization of the sum of the terms becomes difficult.

2.2 Exponential Product Approximation

As we have seen in the previous section, operator exponentiation plays a major role in most fields of physics. For this reason it is necessary to find good approximation to be able to calculate it. The Trotter–Suzuki approximation provides a way to deal with such operations.

The simplest form of the Trotter–Suzuki approximation comes in the following form:

$$\exp\left(x(A+B)\right) = \exp\left(xA\right)\exp\left(xB\right) + O(x^2), \tag{2.10}$$

where A and B are arbitrary general operators with some commutation relation $[A, B] \neq 0$, and x is a parameter. This equation is also known as the Trotter decomposition. To demonstrate that this is actually a first-order approximant, let us rearrange the formula in to following form:

$$\exp\left(xB\right)\exp\left(xA\right) = \exp\left(x(A+B) + O(x^2)\right). \tag{2.11}$$

We can calculate the form of the correction terms that appears in the exponent of the right-hand side by exploiting a Taylor expansion of both sides of Eq. (2.10).

$$\begin{aligned} \exp\left(x(A+B)\right) &= I + x(A+B) + \frac{1}{2}x^2(A+B)^2 + O(x^3) \\ &= I + x(A+B) + \frac{1}{2}x^2(A^2 + AB + BA + B^2) + O(x^3), \end{aligned} \tag{2.12}$$

for the left-hand side, and

$$\begin{aligned}\exp(xA)\exp(xB) &= (I + xA + \frac{1}{2}x^2A^2 + O(x^3))(I + xB + \frac{1}{2}x^2B^2 + O(x^3)) \\ &= I + x(A+B) + \frac{1}{2}x^2(A^2 + 2AB + B^2) + O(x^3)\end{aligned} \tag{2.13}$$

for the right-hand side. The two equations (2.12) and (2.13) differ as the operator A always comes on the left of the operator B in the latter, which let us write the form of the correction term:

$$\exp(xA)\exp(xB) = \exp\left(x(A+B) + \frac{1}{2}x^2[A,B] + O(x^3)\right) \tag{2.14}$$

Therefore, dividing the parameter x into n slices, we get

$$\begin{aligned}\left(\mathrm{e}^{\left(\frac{x}{n}A\right)}\mathrm{e}^{\left(\frac{x}{n}B\right)}\right)^n &= \left[\exp\left(\frac{x}{n}(A+B) + \frac{1}{2}\left(\frac{x}{n}\right)^2[A,B] + O\left(\left(\frac{x}{n}\right)^3\right)\right)\right]^n \\ &= \exp\left(x(A+B) + \frac{1}{2}\left(\frac{x^2}{n}\right)[A,B] + O\left(\left(\frac{x^3}{n^2}\right)\right)\right)\end{aligned}$$

and taking the limit $n \to \infty$ the correction term vanishes, recovering the exponential operator.

It is interesting to compare this approach with another frequently used one, namely the perturbational approximation:

$$\exp(x(A+B)) = I + x(A+B) + O(x^2). \tag{2.15}$$

When dealing with a Hermitian Hamiltonian $H = A+B$, the Trotter–Suzuki approximation has a remarkable advantage over the Eq. (2.15). Indeed, in that scenario, the evolution operator is a unitary operator; the same is not true for the right-hand side of the Eq. (2.15). Contrary, the Trotter–Suzuki preservers this property since it is a product of unitary operators. As a consequence, the norm of the wave function is preserved, resulting in a better accuracy of the evolution. However, a first-order approximation could not be enough to achieve a high precision. For these reasons it is interesting to extend the approximation, looking for higher order approximants.

2.3 Fractal Decomposition

To go beyond the simple approximation presented in the previous section, we can introduce a recursive approach, called fractal decomposition. Bearing

in mind that we want to preserve the unitarity of the approximant, we are looking for an approximation of exponentials products.

The easiest improvement of the Trotter formula (2.10) is the symmetrization

$$S_2(x) \equiv \exp\left(\frac{x}{2}A\right)\exp\left(xB\right)\exp\left(\frac{x}{2}A\right) = \exp\left(f(x)\right). \tag{2.16}$$

The symmetrized approximant has the property

$$\begin{aligned} S_2(-x)S_2(x) =& \exp\left(-\frac{x}{2}A\right)\exp\left(-xB\right)\exp\left(-\frac{x}{2}A\right)\cdot \\ & \exp\left(\frac{x}{2}A\right)\exp\left(xB\right)\exp\left(\frac{x}{2}A\right) = I, \end{aligned}$$

which proves that $f(x)$ does not have an even-order term in x. Consequently, S_2 is a second-order approximant, with the following form

$$S_2 = \exp\left(x(A+B) + x^3R_3 + x^5R_5 + \cdots\right), \tag{2.17}$$

where R_i are suitable operators.

A fourth-order approximant can be constructed from S_2 considering the product

$$\begin{aligned} S(x) =& S_2(sx)S_2((1-2s)x)S_2(sx) && \text{(2.18a)} \\ =& \exp\left(\frac{s}{2}xA\right)\exp\left(sxB\right)\exp\left(\frac{1-s}{2}xA\right)\exp\left((1-2s)xB\right)\cdot \\ & \exp\left(\frac{1-s}{2}xA\right)\exp\left(sxB\right)\exp\left(\frac{s}{2}xA\right), && \text{(2.18b)} \end{aligned}$$

where s is an arbitrary real number. Using Eq. (2.17) the expression (2.18) becomes

$$\begin{aligned} S(x) =& S_2(sx)S_2((1-2s)x)S_2(sx) && \text{(2.19a)} \\ =& \exp\left(sx(A+B) + s^3x^3R_3 + O(x^5)\right)\cdot \\ & \exp\left((1-2s)x(A+B) + (1-2s)^3x^3R_3 + O(x^5)\right)\cdot \\ & \exp\left(sx(A+B) + s^3x^3R_3 + O(x^5)\right) && \text{(2.19b)} \\ =& \exp\left(x(A+B) + (2s^3 + (1-2s)^3)R_3 + O(x^5)\right) && \text{(2.19c)} \end{aligned}$$

The property $S(-x)S(x) = I$ also holds in this case, so we can conclude that the even-order correction in the exponent of (2.19a) will vanish, and the parameter s must satisfy

$$2s^3 + (1-2s)^3 = 0. \tag{2.20}$$

Solving Eq. (2.20), we obtain $s = \frac{1}{2-\sqrt[3]{2}} = 1.351207$. Suppose now that $S(x)$ is an approximation of the time-evolution operator, from time $t = 0$ to $t = x$. The right term $S_2(sx)$ on the right-hand side of Eq. (2.18a) evolves the system from $t = 0$ to $t = sx > x$. The middle-term $S_2((1-2s)x)$ approximates the time evolution from $t = sx$ to $t = sx+(1-2s)x = (1-s)x$. Finally the last term $S_2(sx)$ approximates the evolution from $t = (1 - s)x$ to $t = x$. Representing the evolution as in Fig. 2.1(a), it is evident that the evolution has a part that goes to the "past". In some cases this can be problematic, for instance when studying the diffusion from a delta peak as initial state. Indeed, in this case there is no past of the initial delta peak state.

However, this problem can be easely solved by introducing another fourth-order approximant. Following the same idea, we consider

$$S_4(x) = S_2(s_2x)^2 S_2((1 - 4s_2)x)S_2(s_2x)^2, \tag{2.21}$$

where s_2 is the parameter that solves the equation

$$4s_2^3 + (1 - 4s_2)^3 = 0 \quad \text{or} \quad s_2 = \frac{1}{4 - \sqrt[3]{4}} \simeq 0.4145. \tag{2.22}$$

Similarly to the $S(x)$, we represent $S_2(x)$ as in Fig. 2.2(b). Note that in this case the evolution remains between the initial and final time.

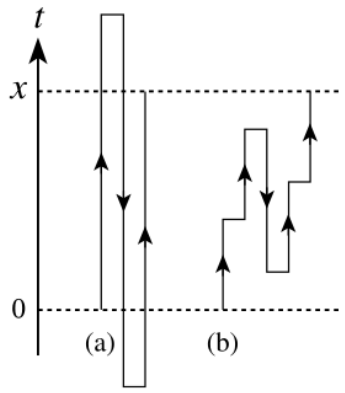


Figure 2.1: Representation of the fourth-order approximation using Eq. (2.18) (Fig. (a)) and Eq. (2.21) (Fig. (b)). In the Trotter–Suzuki approximation, the $2n$-th order approximation of evolution operator, from $t = 0$ to $t = x$ and with $n \geq 1$, is decomposed in a chain of second-order approximants, that evolve the state back and forth in the time.

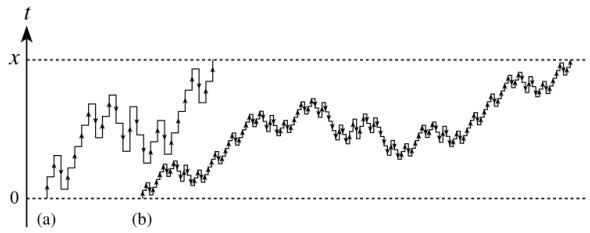


Figure 2.2: Representation of the time evolution using the sixth-order approximant (a) and the eighth-order approximant (b).

Now we can move to the sixth-order approximant, using $S_4(x)$ and following the same structure:

$$S_6(x) = S_4(s_4 x)^2 S_4((1-4s_4)x) S_4(s_4 x)^2, \tag{2.23}$$

obtaining

$$4s_4^5 + (1-4s_4)^5 = 0 \quad \text{or} \quad s_4 = \frac{1}{4 - \sqrt[5]{4}} \simeq 0.3731. \tag{2.24}$$

We can continue this recursive procedure, ending up with the exact time evolution. It is easy to see that the procedure can be generalized with the following formula:

$$S_{2n+2} = S_{2n}(s_{2n} x)^2 S_{2n}((1-4s_{2n})x) S_{2n}(s_{2n} x)^2, \tag{2.25}$$

where $s_{2n} = 1/(4 - 4^{\frac{1}{2n+1}})$. It is interesting to note that the recursive procedure creates a fractal pattern composed by back-and-forth evolution, reproducing the exact time evolution.

2.4 Example: Spin Precession

It is worthwhile to give a brief and simple example to show some remarkable properties of the Trotter–Suzuki decomposition. In this section, we compare it with the first order perturbation, using a simple example of quantum dynamics, namely, the spin precession.

Consider the following Hamiltonian:

$$H = \sigma_z + \Gamma \sigma_x, \tag{2.26}$$

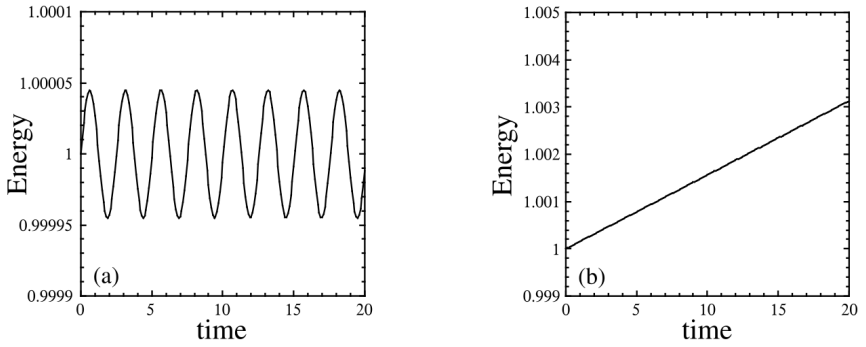


Figure 2.3: Energy expectation value for (a) the Trotter–Suzuki approximation (2.29) and (b) the perturbation approximation (2.30).

where Γ is a real number, and, as initial state, the up-spin state

$$\psi(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \tag{2.27}$$

The evolution is simple to calculate analytically: the spin precesses around the axis of the magnetic field $\vec{H} = (\Gamma, 0, 1)$ with the period

$$T = \frac{\pi}{\sqrt{1+\Gamma^2}}. \tag{2.28}$$

However, here we use the Trotter approximation

$$\exp\left(-\frac{\imath}{\hbar}H\delta t\right) = \exp\left(-\frac{\imath}{\hbar}\sigma_z\delta t\right)\exp\left(-\frac{\imath}{\hbar}\Gamma\sigma_x\delta t\right), \tag{2.29}$$

and the perturbational approximant

$$\exp\left(-\frac{\imath}{\hbar}H\delta t\right) = I - \frac{\imath}{\hbar}(\sigma_z + \Gamma\sigma_x)\delta t, \tag{2.30}$$

Due to the approximations used, the energy expectation $\langle H\rangle$ is not constant throughout the evolution Fig. 2.3(a), as we would expect in the exact solution. However, with the Trotter approximation, the error in the energy expectation oscillates periodically, and never increases beyond the oscillation amplitude. This behavior is consistent with the property of the approximation: due to the unitarity, the state periodically comes back to the initial state with a good accuracy, producing the oscillation pattern in the energy.

In contrast, the error in the energy monotonically grows in the case of the perturbational approximant, as is shown in Fig. 2.3(b). Indeed, with this approximation, the norm of the wave vector increases by the factor

$$\|1 - \frac{\imath}{\hbar} H \Delta t H\| \simeq 1 + \Delta t \|H\| > 1. \tag{2.31}$$

This simple example shows how the unitarity of the Trotter approximation improves the quality of the simulation, compared to the perturbation approximation.

2.5 Example: Symplectic Integrator

Another interesting example regards the study of chaotic dynamics. In this case it is important to keep the symplecticity of the Hamilton dynamics.

Consider a classical Hamiltonian

$$H(\vec{p}, \vec{q}) = K(\vec{p}) + V(\vec{q}), \tag{2.32}$$

where $K(\vec{p})$ is the kinetic term and $V(\vec{q})$ the potential term. The Hamilton equation is expressed in the form

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{pmatrix} \vec{p}(t) \\ \vec{q}(t) \end{pmatrix} = \begin{pmatrix} -\frac{\mathrm{d}}{\mathrm{d}\vec{q}} V(\vec{q}) \\ \frac{\mathrm{d}}{\mathrm{d}\vec{p}} K(\vec{p}) \end{pmatrix} \equiv \begin{pmatrix} & -\hat{V}\cdot \\ \hat{K}\cdot & \end{pmatrix} \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} \tag{2.33}$$

where the operators $\hat{K}\cdot$ and $\hat{V}\cdot$ act in the following way

$$\hat{K} \cdot \vec{p} \equiv \frac{\mathrm{d}}{\mathrm{d}\vec{p}} K(\vec{p}) \quad \text{and} \quad \hat{V} \cdot \vec{q} \equiv \frac{\mathrm{d}}{\mathrm{d}\vec{q}} V(\vec{q}). \tag{2.34}$$

We can define the "Hamiltonian" operator as $\mathcal{H} = \mathcal{K} + \mathcal{V}$ with

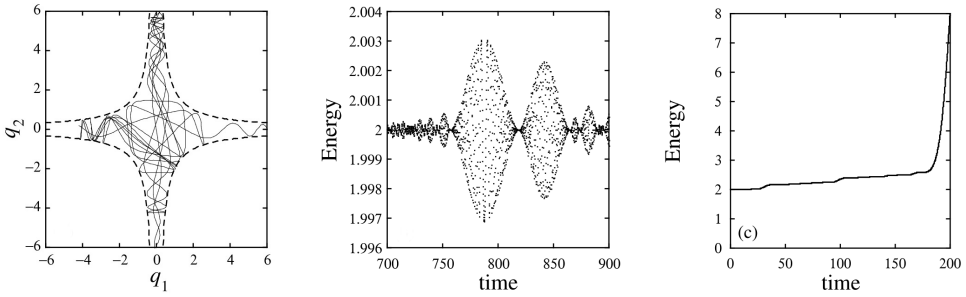
$$\mathcal{K} \equiv \begin{pmatrix} & \\ \hat{K}\cdot & \end{pmatrix} \quad \text{and} \quad \mathcal{V} \equiv \begin{pmatrix} & -\hat{V}\cdot \\ & \end{pmatrix} \tag{2.35}$$

The two operators, $\mathcal{K}$ and $\mathcal{V}$, do not commute. This make the exponential of $\mathcal{H}$ not easily tractable. However, the exponential of $\mathcal{K}$ and $\mathcal{V}$ is easier to calculate.

We notice that $\mathcal{K}^2 = \mathcal{V}^2 = 0$, and therefore we have

$$\exp(\mathcal{K}\Delta t) \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} = (I + \mathcal{K}\Delta t) \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} = \begin{pmatrix} \vec{p} \\ \vec{q} + \Delta t \frac{\mathrm{d}}{\mathrm{d}\vec{p}} K(\vec{p}) \end{pmatrix}, \tag{2.36}$$

$$\exp(\mathcal{V}\Delta t) \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} = (I + \mathcal{V}\Delta t) \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} = \begin{pmatrix} \vec{p} - \Delta t \frac{\mathrm{d}}{\mathrm{d}\vec{q}} V(\vec{q}) \\ \vec{q} \end{pmatrix}. \tag{2.37}$$



(a) (b) (c)

Figure 2.4: (Evolution of the point in the configuration space, using the Trotter approximation (a). The initial condition is $p_1 = p_2 = 0$ and $q_1 = 2$, $q_2 = 1$. The energy fluctuation due to the Trotter approximation (b). The energy increase due to the perturbation approximation (c).

Thus, given the form of $K(\vec{p})$ and $V(\vec{q})$ the Trotter approximation of the evolution operator is determined.

Umeno and Suzuki [42, 40] demonstrated the use of symplectic integrators for chaotic dynamics of the system

$$K(\vec{p}) = \frac{1}{2}(p_1^2 + p_2^2) \quad \text{and} \quad V(\vec{q}) = \frac{1}{2}q_1^2 q_2^2. \tag{2.38}$$

The dynamic is constrained by the constant $\frac{\mathrm{d}}{\mathrm{d}t}|q_1(t)q_2(t)| = 0$, thus tuple $(q_1(t), q_2(t))$ is confined in the area surrounded by four hyperbolas as illustrated in Fig. 2.4a. The Trotter approximation of the dynamics, gives the energy fluctuation shown in Fig. 2.4b. Although the energy deviates from the correct value sometimes, it comes back after the deviation. The major deviations occur when the system goes into one of the four narrow valleys of the potential, while they are suppressed when the system is in the central area.

Contrary to this behaviour, the perturbation approximation

$$\begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} \longrightarrow (I + \Delta t \mathcal{H}) \begin{pmatrix} \vec{p} \\ \vec{q} \end{pmatrix} = \begin{pmatrix} \vec{p} - \Delta t \frac{\mathrm{d}}{\mathrm{d}\vec{q}} V(\vec{q}) \\ \vec{q} + \Delta t \frac{\mathrm{d}}{\mathrm{d}\vec{p}} K(\vec{p}) \end{pmatrix}, \tag{2.39}$$

yields the monotonic energy increase shown in Fig. 2.4c. The reason of the difference between the approximants must be keeping the symplecticity, or the conservation of the phase-space volume.

Chapter 3

Decomposition of Unitary Evolution

When it comes to solving differential equations that determine the behaviour of a physical system, an accurate approximation is often needed to find an explicit solution. As we have seen in the previous chapter, a good approach is to use the Trotter–Suzuki approximation.

In this chapter we are interested in calculating the evolution operator $U(t) = \exp\left(-\frac{it}{\hbar}H\right)$, the solution of the time-dependent Schrödinger equation, using the second order Trotter–Suzuki approximation. The Trotter–Suzuki approximation is based on the splitting of the exponent in a sum of operators. Thus, starting from a exponential that is difficult to calculate, we end up with a product of easy-to-calculate exponentials. In the next sections, we discuss how to decompose the Hamiltonian and the eventual calculation of the exponential.

3.1 Hamiltonian Decomposition

We consider a single quantum particle in two dimensions in time-independent potential. The Hamiltonian operator of such system is written as follows

$$\hat{H} = \frac{\hat{P}_x^2 + \hat{P}_y^2}{2m} + \hat{V}, \tag{3.1}$$

where m is the particle mass and $\hat{V}$ is the external potential.

We use the coordinate reppresentation of the operators, so the kinetic

term becomes

$$\langle x,y|\,\frac{1}{2m}\left(P_x^2+P_y^2\right)|\psi\rangle = \int \mathrm{d}x'\mathrm{d}y'\,\langle x,y|\left(P_x^2+P_y^2\right)|x',y'\rangle\,\langle x',y'|\psi\rangle \quad (3.2a)$$

$$= -\frac{\hbar^2}{2m}\left(\nabla_x^2+\nabla_y^2\right)\psi(x,y) \quad (3.2b)$$

In this basis, the exponentiation of the external potential operator is straightforward, since it is diagonal. On the contrary, this is not true for the kinetic operator.

We consider the discretization of the continuum space into a uniform mesh, where Δ is the distance between any two consecutive points. We use the tuple (i,j) to label the points of the mesh, with $i,j=1,\ldots,N$, so that $\psi(x,y)\to\psi_{i,j}$ and $|x,y\rangle\to|i,j\rangle$. Using the second-order derivative central difference, we have:

$$\left.\frac{\partial^2\psi}{\partial x^2}\right|_{i,j} = \frac{\psi(i+1,j)-2\psi(i,j)+\psi(i-1,j)}{\Delta^2}+O(\Delta^2), \quad (3.3)$$

Then we can write the Eq. (3.2b) as

$$\langle i,j|\,\frac{1}{2m}\left(P_x^2+P_y^2\right)|\psi\rangle = -\frac{\hbar^2}{2m\Delta^2}\left(\psi_{i+1,j}+\psi_{i,j+1}+\psi_{i-1,j}+\psi_{i,j-1}-4\psi_{i,j}\right).$$

To explicitly determine the matrix elements of the kinetic operator, let us rewrite the previous equation using Kroneker's delta

$$\langle i,j|\,\frac{1}{2m}\left(P_x^2+P_y^2\right)|\psi\rangle = \sum_{k,l} -\frac{\hbar^2}{2m\Delta^2}[(\delta_{i+1,k}+\delta_{i-1,k})\,\delta_{j,l}+ \quad (3.4)$$

$$+(\delta_{j+1,l}+\delta_{j-1,l})\,\delta_{i,k}-4\delta_{i,k}\delta_{j,l}]\psi_{k,l}.$$

Since the discretization of Eq. (3.2a) led to the following equation

$$\langle i,j|\,\frac{1}{2m}\left(P_x^2+P_y^2\right)|\psi\rangle = \frac{1}{2m}\sum_{k,l}\langle i,j|\left(P_x^2+P_y^2\right)|k,l\rangle\,\langle k,l|\psi\rangle \quad (3.5a)$$

$$= \frac{1}{2m}\sum_{k,l}\langle i,j|\left(P_x^2+P_y^2\right)|k,l\rangle\,\psi_{k,l} \quad (3.5b)$$

from the comparison of Eq. (3.4) and Eq. (3.5b), we get

$$\langle i,j|\,\frac{1}{2m}\left(P_x^2+P_y^2\right)|k,l\rangle = -\frac{\hbar^2}{2m\Delta^2}[(\delta_{i+1,k}+\delta_{i-1,k})\,\delta_{j,l}+ \quad (3.6)$$

$$+(\delta_{j+1,l}+\delta_{j-1,l})\,\delta_{i,k}-4\delta_{i,k}\delta_{j,l}].$$

We introduce two operators that will let us split the Hamiltonian into a sum of operators that are easy to exponentiate. We define:

$$A_{i,k} = \begin{cases} \delta_{i+1,k}, & \text{if } k \text{ is odd} \\ \delta_{i-1,k}, & \text{if } k \text{ is even} \end{cases} \tag{3.7}$$

and

$$B_{j,l} = \begin{cases} \delta_{j-1,l}, & \text{if } l \text{ is odd} \\ \delta_{j+1,l}, & \text{if } l \text{ is even} \end{cases} \tag{3.8}$$

Represented as matrices, these operators have the form of block diagonal matrices, namely:

$$A = \begin{pmatrix} 0 & 1 & & & & & \\ 1 & 0 & & & & & \\ & & 0 & 1 & & & \\ & & 1 & 0 & & & \\ & & & & 0 & 1 & \\ & & & & 1 & 0 & \\ & & & & & & \ddots \end{pmatrix} \quad B = \begin{pmatrix} 0 & & & & & & \\ & 0 & 1 & & & & \\ & 1 & 0 & & & & \\ & & & 0 & 1 & & \\ & & & 1 & 0 & & \\ & & & & & 0 & 1 \\ & & & & & 1 & 0 \\ & & & & & & \ddots \end{pmatrix} \tag{3.9}$$

Using the new operators we can rewrite Eq. (3.6) as follow

$$\begin{aligned} \langle i,j| \frac{1}{2m}\left(P_x^2 + P_y^2\right)|k,l\rangle = -\frac{\hbar^2}{2m\Delta^2}[\,&(A_{i,k} + B_{i,k})\,\delta_{j,l} + \\ &+ (A_{j,l} + B_{j,l})\,\delta_{i,k} - 4\delta_{i,k}\delta_{j,l}] \end{aligned} \tag{3.10}$$

For the brevity of notation, we adopt the operator notation, so that the previous equation becomes

$$\frac{1}{2m}\left(\hat{P}_x^2 + \hat{P}_y^2\right) = -\frac{\hbar^2}{2m\Delta^2}\left[\hat{A}_x + \hat{B}_x + \hat{A}_y + \hat{B}_y - 4\hat{I}\right], \tag{3.11}$$

where the label indicates the index on which the operator acts, so that the following commutation rules are satisfied:

$$[\hat{A}_x, \hat{A}_y] = 0 \quad [\hat{A}_x, \hat{B}_y] = 0 \quad [\hat{B}_x, \hat{A}_y] = 0 \quad [\hat{B}_x, \hat{B}_y] = 0. \tag{3.12}$$

Finally, by Eq. (3.11), we get the decomposition formula for the Hamiltonian (3.1):

$$\hat{H} = -\frac{\hbar^2}{2m\Delta^2}\left[\hat{A}_x + \hat{B}_x + \hat{A}_y + \hat{B}_y - 4\hat{I}\right] + \hat{V}. \tag{3.13}$$

3.2 Evolution Operator

In the previous section, we shown how to split the Hamiltonian in the discrete space approximation. Now we explicitly calculate the Trotter–Suzuki decomposition for the evolution operator. Using the Hamiltonian decomposition (3.13), the evolution operator $\hat{U}(t) = \exp(-\frac{\imath t}{\hbar}H)$ can be written as follows, in the first Trotter–Suzuki approximation

$$\hat{U}_1(t) = \exp\left(-\frac{\imath t}{\hbar}\left(\hat{V} + \frac{2\hbar^2}{m\Delta^2}\hat{I}\right)\right)\exp\left(\imath\alpha\hat{A}_x\right)\exp\left(\imath\alpha\hat{B}_x\right)\cdot \\ \cdot\exp\left(\imath\alpha\hat{A}_y\right)\exp\left(\imath\alpha\hat{B}_y\right) + O(t^2), \tag{3.14}$$

where we defined $\alpha = \frac{\hbar t}{2m\Delta^2}$.

Using the equality

$$\exp(\imath\alpha\sigma) = I\cos(\alpha) + \imath\sigma\sin(\alpha), \tag{3.15}$$

it is straightforward to calculate the exponential of the operators A and B, since they are diagonal matrices of the Pauli matrix:

$$\exp\left(\imath\alpha\hat{A}_x\right) = \hat{I}_x\cos(\alpha) + \imath\sin(\alpha)\hat{A}_x \tag{3.16}$$

$$\exp\left(\imath\alpha\hat{B}_x\right) = \hat{I}_x(\cos(\alpha)(1-\delta_{x,0}) + \delta_{x,0}) + \imath\sin(\alpha)\hat{B}_x. \tag{3.17}$$

The first exponential in Eq. (3.14) is also straightforward to calculate. Indeed $\hat{V}\,|i,j\rangle = V(i,j)\,|i,j\rangle$ so,

$$\langle k,l|\exp\left(-\frac{\imath t}{\hbar}\left(\hat{V} + \frac{2\hbar^2}{m\Delta^2}\hat{I}\right)\right)|i,j\rangle = \delta_{k,i}\delta_{l,j}\exp\left(-\frac{\imath t}{\hbar}\left(V(i,j) + \frac{2\hbar^2}{m\Delta^2}\right)\right).$$

The second order approximation of Trotter–Suzuki decomposition is easily calculated using $U_1(t)$, namely

$$\hat{U}_2(t) = \hat{U}_1\left(-\frac{t}{2}\right)^{\dagger}\hat{U}_1\left(\frac{t}{2}\right). \tag{3.18}$$

3.3 Evolution Towards the Ground-state

A reliable and easily-implemented method of approximating the ground state of the system is by propagation in imaginary time. Consider the Schrödinger equation

$$\imath\hbar\frac{\partial\,|\psi(t)\rangle}{\partial t} = \hat{H}\,|\psi(t)\rangle\,. \tag{3.19}$$

The transformation $\tau = \imath t$ lead to the equation

$$\hbar \frac{\partial \left|\psi(\tau)\right\rangle}{\partial \tau} = -\hat{H} \left|\psi(\tau)\right\rangle . \tag{3.20}$$

The formal solution for this equation, with initial condition $|\psi(0)\rangle = |\psi_0\rangle$ is $|\psi(\tau)\rangle = \exp\left(-\frac{\tau}{\hbar}H\right)|\psi_0\rangle$. The initial state $|\psi_0\rangle$ can be written as a linear combination of Hamiltonian's eingenvectors:

$$\left|\psi_0\right\rangle = \sum_i c_i \left|\phi_i\right\rangle , \tag{3.21}$$

where $\hat{H}\left|\phi_i\right\rangle = E_i\phi_i$ for $i = 0, 1, 2, \ldots$. In this basis $|\psi(\tau)\rangle$ can be written as follow

$$\left|\psi(\tau)\right\rangle = \sum_i c_i \exp\left(-\frac{\tau}{\hbar}E_i\right)\left|\phi_i\right\rangle . \tag{3.22}$$

Taking E_0 as the ground-state energy, we can rearrange the previous equation

$$\left|\psi(\tau)\right\rangle = \exp\left(-\frac{\tau}{\hbar}E_0\right)\sum_i c_i \exp\left(-\frac{\tau}{\hbar}\Delta E_i\right)\left|\phi_i\right\rangle . \tag{3.23}$$

where $\Delta E_i = E_i - E_0 > 0, \forall i > 0$. We now take the limit for $\tau \to +\infty$. As long as the initial state is not orthogonal to the ground state, namely $c_0 \neq 0$, the leading term in the sum of Eq. (3.23) is given by the ground state

$$\lim_{\tau\to+\infty}\left|\psi(\tau)\right\rangle = \exp\left(-\frac{\tau}{\hbar}E_0\right)c_0\left|\phi_0\right\rangle . \tag{3.24}$$

Thus, evolving the initial state for a sufficient amount of time, it let us reach an approximation of the ground state.

We implement the evolution operator in imaginary time using the same Trotter–Suzuki decomposition and Hamiltonian splitting as for the real time evolution. The operator reads as

$$\hat{U}(\tau) = \exp\left(-\frac{\tau}{\hbar}\left(-\frac{\hbar^2}{2m\Delta^2}\left[\hat{A}_x + \hat{B}_x + \hat{A}_y + \hat{B}_y - 4\hat{I}\right] + \hat{V}\right)\right), \tag{3.25}$$

so in the first Trotter–Suzuki approximation we have

$$\begin{aligned}\hat{U}_1(t) = \exp\left(-\frac{\tau}{\hbar}\left(\hat{V} + \frac{2\hbar^2}{m\Delta^2}\hat{I}\right)\right)\exp\left(\alpha_\tau\hat{A}_x\right)\exp\left(\alpha_\tau\hat{B}_x\right)\cdot \\ \cdot\exp\left(\alpha_\tau\hat{A}_y\right)\exp\left(\alpha_\tau\hat{B}_y\right),\end{aligned} \tag{3.26}$$

where $\alpha_\tau = \frac{\hbar\tau}{2m\Delta^2}$. Using the equality

$$\exp(\alpha_\tau \sigma) = I\cosh(\alpha_\tau) + \sigma\sinh(\alpha_\tau) \tag{3.27}$$

we can calculate the exponential of A and B as

$$\exp\left(\alpha_\tau A_x\right) = I_x\cosh(\alpha_\tau) + \sinh(\alpha_\tau)A_x \tag{3.28}$$

$$\exp\left(\alpha_\tau B_x\right) = I_x(\cosh(\alpha_\tau)(1-\delta_{x,0}) + \delta_{x,0}) + \sinh(\alpha_\tau)B_x \tag{3.29}$$

Finally, in the second order approximation we have:

$$\hat{U}_2(\tau) = \hat{U}_1\left(\frac{\tau}{2}\right)^{\mathrm{T}}\hat{U}_1\left(\frac{\tau}{2}\right) \tag{3.30}$$

Chapter 4

Implementation

Our main goal was to develop a high-performance algorithm. The implementation uses a distributed version of highly optimized kernel for central processing units (CPUs) and graphics processing units (GPUs), that runs efficiently on a cluster [43]. We extended this implementation to Hamiltonians that include external potential, to allow the simulation of a wider range of quantum systems.

Particularly important for the purpose of high performance is the optimization of memory access patterns. Large amounts of data are stored in the main memory: the data needs to be sent to the processing unit. Nowadays, processing units are much faster in performing calculations than the ability of the main memories and the hardware bus to keep streaming data. So if the processing unit has to fetch data from the main memory, it would be limited by the bandwidth of the memory and the hardware bus. To avoid this problem, we exploit cache-aware computation that uses smaller and faster memories, dubbed caches [10]. Furthermore, a workload across a distributed memory system requires communication between the nodes. To proceed to the next iteration, a node needs data of the previous iteration calculated by other nodes. The transfer of data in a network of nodes is even slower than the transfer from main memory to the processing unit. However, to a certain extent, communication between nodes and calculation in the processing unit can be done simultaneously. For the sake of efficiency, it is worth to overlap the two as much as possible.

For the purpose of developing reliable scientific software, we added unit testing to the implementation. In the development of complex software, it is important to test various parts of the code, to ensure its correctness. A program can be split into several units, each one having a defined use and

an expected behaviour. Based on this, one can develop a test to exercise the unit and verify its exactness. We exploit unit testing using the library CppUnit [20]. Moreover, we use double-precision floating point operations, to improve accuracy in the simulations.

Our approach was also to ensure that our implementation fits in with rapid prototyping systems [29]. The program allows to the use of a command line interface, for the flexibility of the simulation. In addition, the function that performs the evolution is exposed as an application programming interface (API). We also developed wrappers to make the kernels accessible from the high-level languages Python and MATLAB.

In this chapter, we describe the implementation of the evolution operator. There are two CPU kernels, one GPU kernel and a hybrid kernel that use both types of computational units. Before the kernel explanation, a short section introduces to the architecture of CPU and GPU memory hierarchy and gives some basic concepts of high performance programming. We end with the benchmark performed at the Barcelona Supercomputing Center.

4.1 Cache Optimization

The gap between CPU speed and main memory performance is enormous. To alleviate this gap, computer architectures implement hierarchical memory structures. This approach allows to work around both the low main *memory bandwidth* and the *latency* of main memory accesses. The memory bandwidth is a measure of the rate at which data can be read from or stored into the memory by a processor, and it is a crucial parameter that affects the performance of an algorithm. Furthermore, the memory latency plays an important role in the overall performance. The memory latency is the delay time between the moment a memory controller tells the memory module to access a particular memory location, and the moment these data become available on the module's output pins. These parameters characterized the velocity at which the memory can feed the processor. When the CPUs need to process a certain data, they request it from the memory, and wait for it to become available.

The common structure of the hierarchy consists of a series of memories; the smaller they are, the closer they are to the CPUs; the cheaper they are, the further they are from the CPUs. Usually, at the top of the hierarchy there are the registers, memories integrated within the processor chip that can provide data with low latency and high bandwidth. Between the pro-

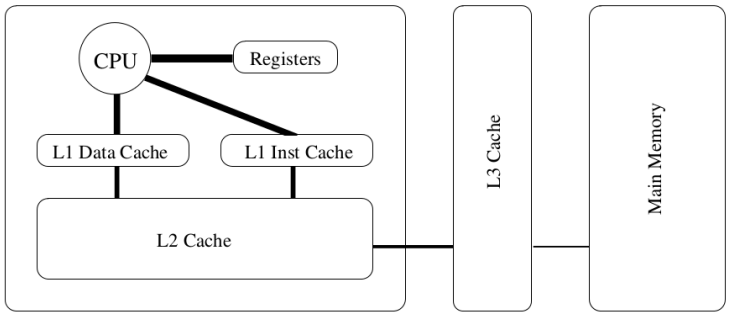


Figure 4.1: A Common memory hierarchy that present two on-chip L1 caches, on-chip L2 cache, and a third level of off-chip cache. The thickness of the interconnections illustrate the bandwidths between the memory hierarchy levels.

cessor core and the main memory there are memories called *cache memories* or *caches* [10]. Finally, there is the main memory, that usually consist of large and slow RAM memories. During the execution of a program, some blocks of data are used more often than others, so the CPUs will work with this subset of data for most of the time. To get an efficient algorithm, the idea is to store frequently used blocks of data on fast memories: the more frequently the block is used, the higher in the hierarchy is the memory that stores it.

Typically, the data residing within a smaller memory are also stored within the larger memory, so the levels of the memory hierarchy are subsets of one another. A common memory hierarchy is show in Fig. 4.1.

An efficient algorithm must consider the stages of the memory hierarchy. Unfortunately, compilers are not intended to introduce sophisticated cache-based transformations. Consequently, the optimization effort is left to the programmer.

This aspect is particularly important when dealing with numerically intense codes, which occur in science and engineering disciplines, such as computational physics, mechanical engineering and computational fluid dynamics, just to mention some. These types of code are characterized by a large portion of floating-point operations, and small computational kernels. Thus, instruction cache misses do not significantly affect the execution performance. Much of the optimization effort concerns data access pattern. Indeed, due to data access latencies and memory bandwidth issues, it is not sufficient to optimize the number of arithmetic operation alone. Efficient

codes in scientific computing must necessarily combine computationally optimal algorithms and memory hierarchy optimization.

4.1.1 Organization of Cache Architectures

The common memory hierarchy presents a rather small number of registers on the chip, which has almost no memory latency. On the chip we can also find a small cache - called *level one (L1) cache* – usually limited to 64 Kbyte, so that low latency and high bandwidth are assured. The latency of on-chip caches is commonly one or two CPUs cycles. The L1 cache is often split into two separate parts; one only keeps data, the other instructions. The second level memory (L2) is tipically placed on-chip as well and it is usually limited to 1 Mbyte. Due to the bigger size the latency is around 5 to 10 cycles. Another cache level may be implemented off-chip if the L2 cache is on-chip. The L3 cache size may vary from 1 MByte to 16 MByte. They provide data with latency of about 10 to 20 cycles [12].

Data within the cache are stored in *cache lines*. A cache line holds the contents of a contiguous block of main memory. We say that a cache hit occurs when the data requested by the processor is found in a cache line. If the data requested is not founded in the L1 cache, a cache miss occurs. In the latter case, the contents of the memory block containing the requested words are then fetched from a lower memory layer, for instance, from the L2 cache, and copied into a cache line. This operation typically implies another chunk data in L1 to be replaced by the requested one – an operation that is very inefficient. Indeed, the replacement of a cache line takes more time than the CPU to read the same data directly from the main memory. For this reason, caches implement strategies to increase the rate of cache hits over the cache misses. The optimal replacement strategy would be to replace the memory block which will not be accessed for the longest time. However, such strategy is impossible to implement since it requires information about future cache references. The most commonly used strategy is the *least recently used*. It replaces the block which has not been accessed for the longest time interval.

These strategy are based on the principle of locality references [12], which states that recently used data are very likely to be reused in the near future. Locality can be of two different type: temporal locality and spatial locality. A sequence of references exhibits temporal locality if recently accessed data are likely to be accessed again in the near future. A sequence of references manifest spatial locality if data located close together in address space tend to be referred close together in time.

4.1.2 Data Access Optimizations

The most straightforward and simple approach to implement an algorithm, usually does not achieve the best execution performance. As we saw in the previous section, to reach this goal the programmer has to care about how the data movements are handled by the memory hierarchy and how the CPUs access data. In scientific computations, this typically implies the need to apply a transformation on the code, that change the order in which iterations in a loop nest are executed. Such transformations are part of data access optimizations techniques, where the goal is to improve temporal locality. We focus on a set of loop transformations that improve data locality for one level of the memory hierarchy: a cache.

Loop Interchange. This transformation reverses the order of two adjacent loops in a loop nest. This can be generalized to loop permutations where more than two loops are moved at once [17, 44].

A loop interchange can improve locality by reducing the *stride* of an array-based computation. The stride is the distance of array elements in memory accessed within consecutive loop iterations. For instance, suppose we want to calculate the square norm of a vector (Fig. 4.3). Furthermore, suppose that the vector is stored in a 6 by 8 array in memory (Fig. 4.2), in *row major order*; that is, two array elements are stored adjacent in memory if their second indices are consecutive numbers. The code corresponding to the left part of Fig. 4.2, accesses the array elements in a column-wise manner, so the stride is equal to 8. Consequently, the preloaded data in the cache line marked with grey color will not be reused if the array is too large to fit entirely in cache. The next element will be fetched from the main memory. Interchanging the loop nest allows the cache line to be reused, as the stride is now 1 (right part of Fig. 4.2).

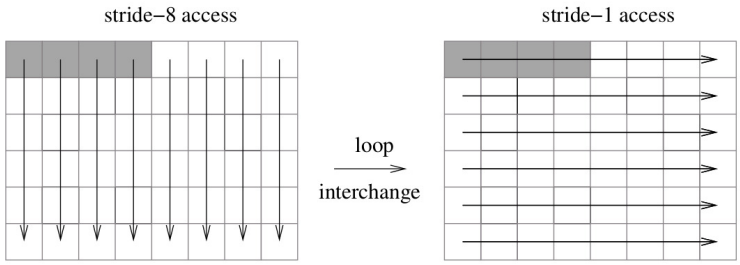


Figure 4.2: Access pattern for interchenged loop nests in a (6,8) array.

```
double norm2;
double a[n, n];
for j = 1 to n do
    for i = 1 to n do
        norm2+ = a[i, j] · a[i, j];
    end
end
```
Algorithm 1: Original loop

```
double norm2;
double a[n, n];
for i = 1 to n do
    for j = 1 to n do
        norm2+ = a[i, j] · a[i, j];
    end
end
```
Algorithm 2: Loop interchange

Figure 4.3: Pseudocode that illustrate the loop interchange transformation.

Loop Fusion. This transformation takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop [6]. When loop fusion can be applied – when there are no instruction dependencies between the fused loops – data locality may be improved. Assume that two consecutive loops perform global sweeps through an array as in the code shown in Fig. 4.4, and that the data are too large to fit entirely in the cache. When the first loop finishes, the elements of array b are not completely loaded in cache, and the second loop will have to reload them from the main memory. If the two loops are combined with loop fusion only one global sweep through the array b will be performed, resulting in fewer cache misses.

```
for i = 1 to n do
    b[i] = a[i] + 1;
end
for i = 1 to n do
    c[i] = b[i] · 3;
end
```
Algorithm 3: Original loop

```
for i = 1 to n do
    b[i] = a[i] + 1;
    c[i] = b[i] · 3;
end
```
Algorithm 4: Loop fusion

Figure 4.4: Pseudocode that illustrate the loop fusion transformation.

4.2 CPU Kernels

The code implements two CPU kernels: both are cache optimized, but one is further optimized to use the SSE instruction set of the CPU. In this

section we explain the CPU kernels in a single thread scenario and the cache optimization strategy adopted.

4.2.1 Matrix Updating Scheme

The initial wave function of the system $\psi_{i,j}(0)$ is stored in two arrays in row major order, one for the real part and one for the imaginary part. The evolved wave function $\psi_{i,j}(t)$ is calculated dividing the time in small time intervals of length Δt. To have an accurate simulation, Δt must satisfy the inequality $\Delta t \ll \frac{m\Delta L^2}{\hbar}$, where m is the particle mass and ΔL is the distance between two consecutive locations in the mesh. Indeed if we exploit the Taylor expansion of the evolution operator

$$\exp\left(\frac{\imath}{\hbar}H\Delta t\right) = 1 + \frac{\imath}{\hbar}H\Delta t + O(\Delta t^2), \tag{4.1}$$

we have $1 \gg \frac{\Delta t}{\hbar}\|H\|$. Suppose that the leading term in the Hamiltonian is the kinetic term, hence the second-order derivative approximation (Eq. (3.3)) leads to $1 \gg \frac{\hbar}{m\Delta L^2}\Delta t$.

In the second-order Trotter–Suzuki approximation, the $\psi_{i,j}(\Delta t)$ is the result of nine matrix-vector products. From Eq. (3.18) and Eq. (3.14), we have

$$\psi(\Delta t) = \mathrm{e}^{\imath\frac{\alpha}{2}\hat{A}_y}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{A}_x}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{B}_y}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{B}_x}\mathrm{e}^{-\frac{\imath\Delta t}{\hbar}\hat{V}}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{B}_x}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{B}_y}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{A}_x}\mathrm{e}^{\imath\frac{\alpha}{2}\hat{A}_y}\psi(0) \tag{4.2}$$

where $\alpha = \frac{\hbar\Delta t}{2m\Delta L^2}$. Note that we discarded the phase changing factor $\mathrm{e}^{-\frac{\imath\Delta t\hbar}{m\Delta L^2}I}$ present in Eq. (3.14). The exponential of the potential is diagonal and it is straightforward to implement: it is sufficient to multiply each element of the vector $\psi_{i,j}$ for the proper value (i.e. $\langle i,j|\,\mathrm{e}^{-\frac{\imath t}{\hbar}\hat{V}}\,|i,j\rangle$). The matrix elements $\langle i,j|\,\mathrm{e}^{-\frac{\imath t}{\hbar}\hat{V}}\,|i,j\rangle$ are calculated before the beginning of the kernel, and is stored on two real matrices in the main memory. With regards to the other exponential operators, the matrix vector multiplication can be decomposed in a series of linear combinations of couples of two neighbouring sites in the mesh. This can be easily understood from the form of A and B operators (Eq. (3.7) and Eq. (3.8)) that lead to Eq. (3.16) and Eq. (3.17). Consider the couple $\psi_{i,j}$ and $\psi_{i,j+1}$, when j is even, $\mathrm{e}^{\imath\frac{\alpha}{2}A_y}$ acts so that

$$\begin{aligned}\hat{\psi}_{i,j} &= \cos\left(\frac{\alpha}{2}\right)\psi_{i,j} + \imath\sin\left(\frac{\alpha}{2}\right)\psi_{i,j+1}\\ \hat{\psi}_{i,j+1} &= \imath\sin\left(\frac{\alpha}{2}\right)\psi_{i,j} + \cos\left(\frac{\alpha}{2}\right)\psi_{i,j+1}\end{aligned} \tag{4.3}$$

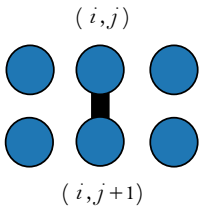


Figure 4.5: Single coupling operation.

In this operation, the sites (i, j) and $(i, j+1)$ in the arrays are updated. We schematically represent this operation by mean of the diagram in Fig. 4.5. Note that we do not need to store the exponential operator matrices $e^{\imath\frac{\alpha}{2}A}$ and $e^{\imath\frac{\alpha}{2}B}$. It is sufficient to store only two values: $\cos\left(\frac{\alpha}{2}\right)$ and $\sin\left(\frac{\alpha}{2}\right)$.

Following this scheme, Eq. (4.2) can be schematized as in Fig. 4.6. The single time evolution is divided in nine computational steps corresponding to the matrix vector multiplications. Furthermore, the operations are independent from one another, so that sites in each computational step can be updated in place.

Regarding the imaginary time evolution, Eq. (4.3) changes into:

$$\begin{aligned}\hat{\psi}_{i,j} &= \cosh\left(\frac{\alpha}{2}\right)\psi_{i,j} + \sinh\left(\frac{\alpha}{2}\right)\psi_{i,j+1} \\ \hat{\psi}_{i,j+1} &= \sinh\left(\frac{\alpha}{2}\right)\psi_{i,j} + \cosh\left(\frac{\alpha}{2}\right)\psi_{i,j+1}\end{aligned} \tag{4.4}$$

4.2.2 Cache-aware Implementation

A naive approach to implement the scheme in Fig. 4.6 is by performing each computational step in a single pass over the entire array of sites. This performs efficiently as long as the array of sites fits in the cache, so that the CPU fetches the data directly from the cache. However, for large system sizes, data need to be fetched from the main memory, resulting in a drop in performance [1].

Cache optimization can be achieved dividing the array of sites in blocks that fit in the cache and performing a single time step evolution for each one of them, separately. This raises the problem of data dependency. Suppose that a block has been read to the cache and evolved a number of steps. We cannot write the results back to the same array in the main memory, from

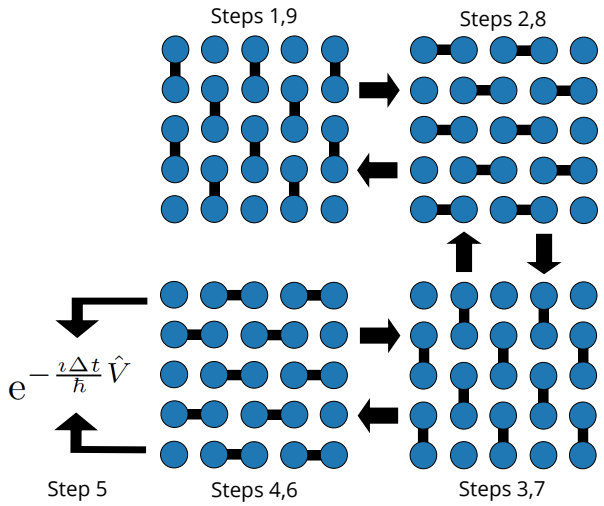


Figure 4.6: Single time step evolution scheme for the second order Trotter–Suzuki.

where the block was read, as blocks adjacent to it still need some values on the boundary between the blocks for their own evolution. This is fixed through double buffering, allocating two arrays in the main memory instead of one and going back and forth between the two, reading from one and writing to the other.

Besides, to perform some computational steps on a block, we need the sites surrounding the block to be present in the cache as well, otherwise the sites on the edge of the block will not be valid. This is because each site needs their four neighbours to complete a single time step evolution. As we try to perform more computational steps on a block, the amount of nodes external to the block increases. These nodes generate a *halo* around a block that must also be read into the cache and updated, but that at later steps become invalid because their own halos are not present. The minimum value of the halo thickness for a single time step evolution is of four sites, since there are four computational steps that couple sites for each direction.

In a non distributed version of the kernels a single CPU performs the time step evolutions as schematized in Fig. 4.6. Blocks of the array in the main memory, with their own halo, are written in the cache memory. A single time step evolution of the block is performed, writing the results back to the cache. The halo is discarded, while the block is written to the second buffer in the main memory (Fig. 4.7). The multiple step strategy, combined

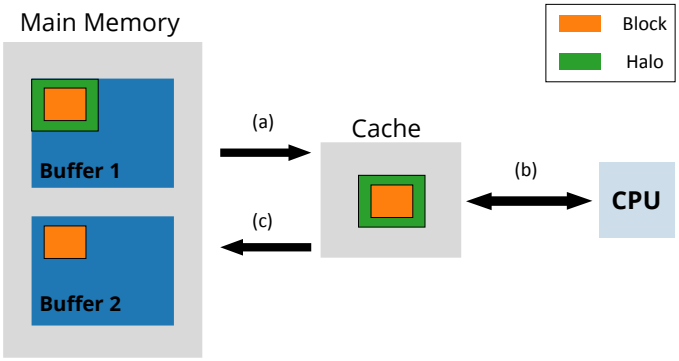


Figure 4.7: Scheme of the CPU cache optimization. A time step evolution is performed: a block, in buffer 1, and its halo are written into the cache (a); a time step evolution is performed on the block and the halo in the cache, by the CPU (b); the halo is discarded and only the block is written into the main memory in buffer 2 (c). This operation is performed for each block in buffer 1.

with the tunable block size that depends on the hardware's cache size, puts the algorithm in the family of cache-aware stencil computations [15].

4.3 GPU Kernel

As the time step evolution is composed by simple and high parallelizable instructions, an implementation that runs on a GPU gains a high speed-up compare to a CPU kernel. Indeed, GPUs achieve a much high peak performances in certain parallel workloads compared to CPUs, as illustrated in Fig. 4.8. This motivated the development of a GPU kernel. In this section, we outline the differences between the CPU and GPU architectures, describing the features of the latter. A description of the non-distributed version of GPU kernel follows.

4.3.1 GPU Structure

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intense, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. In-

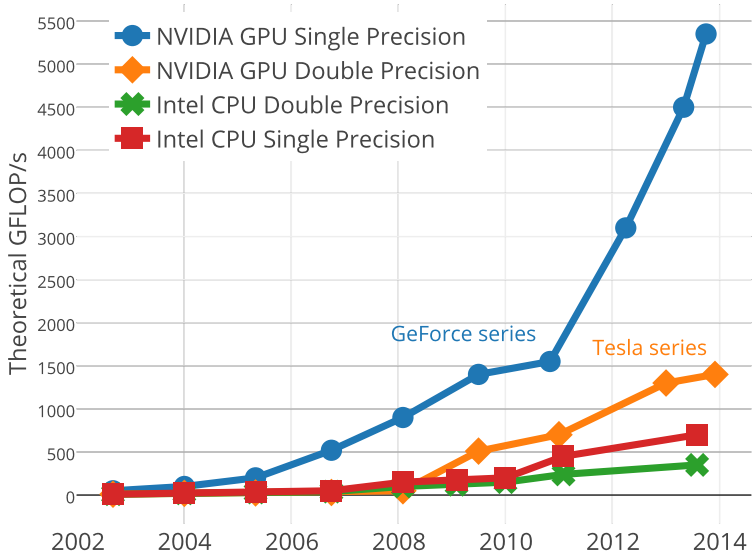


Figure 4.8: Floating-point operations per second for the CPU and GPU.

deed, a GPU is well-suited to address problems that can be expressed as data-parallel computations, where the same calculation is executed on many data elements in parallel. Consequently, a sophisticated flow control mechanism is unnecessary. Furthermore, because the program is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of large data caches.

GPUs adopt the so-called *single instruction, multiple thread* architecture as parallel execution model. Data elements are processed by sequences of parallel instructions called *threads*. The NVIDIA GPU architecture is built around a scalable array of multithreaded *Streaming Multiprocessors*. Threads are managed, created, scheduled and executed by streaming multiprocessors in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are

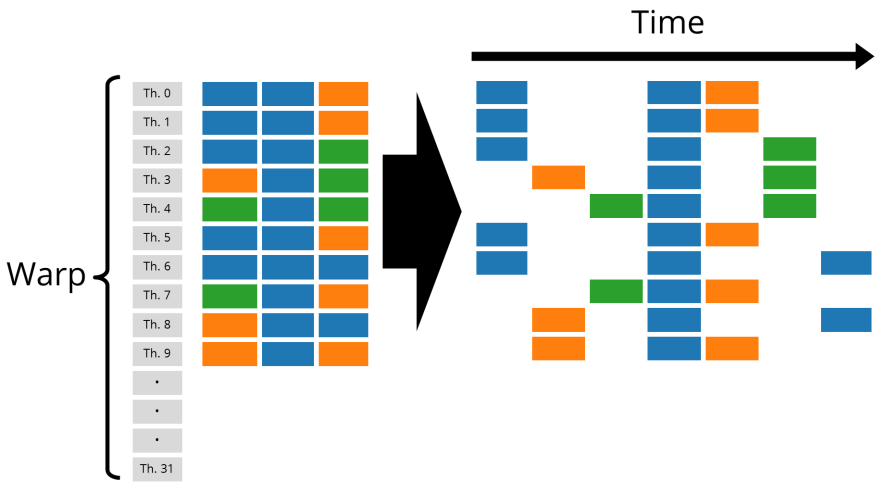


Figure 4.9: Instance of a warp execution. The left part of the graph illustrates the instructions to be performed by each thread. Each thread execute three instructions; instructions with the same color are identical. The right part of the graph illustrates how the instructions are executed by the threads within a warp. Threads, that execute the same instruction, perform the instruction at the same time. Different instructions are performed on different times. In this example threads are not in the same execution path – there are at least three different paths – resulting in a inefficient time performance.

not on that path, and when all paths complete, the threads converge back to the same execution path (Fig. 4.9). Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. Consequently, the execution achieve better performance avoiding threads in a warp to branch.

GPU are easily programmed using CUDA (Compute Unified Device Architecture) an extension of high-level programming language C. CUDA extends C by allowing the programmer to define C functions (kernels) that are executed N times in parallel by N different *CUDA threads*. Each thread is associated to a 3-component vector, so that they can be organized in a 1D, 2D or 3D structure. Groups of threads are collected in *thread blocks*, whose size is dictated by the programmer. However, there is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. Blocks are organized into a *grid* of thread blocks, which can be

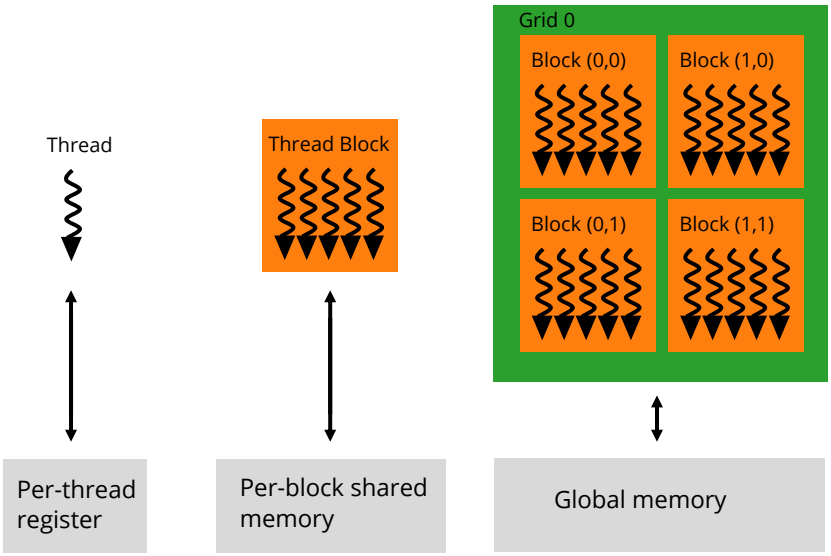


Figure 4.10: GPU memory hierarchy.

one, two or three dimensional.

A GPU's memory is organized in a hierarchy, as it is also the case for the CPU (Fig. 4.10). Closest to the processors there are fast and small registers, in which each thread store a private local memory space. On the next level in the hierarchy, there are *shared memories*. Each thread block has access to a memory space stored on shared memories, which is visible to all the threads of the block. The last level consist in the global memory, accessible by all the threads.

4.3.2 GPU Implementation

The GPU kernel implements the same time step evolution as the CPU kernel (Fig. 4.6). In analogy with the CPU kernel, the GPU kernel divides the array of sites in blocks, which are written in the shared memories of the multiprocessors, along with their own halo. This strategy benefits from the higher bandwidth of the shared memories, as it happens for the CPU cache. Double buffering is required for data dependency; two arrays in the global memory are allocated, one for reading and one for writing the results. Contrary to what happens for the CPU kernels, blocks are executed in parallel, reducing the time of the execution.

The time step evolution is performed using half number of threads as

there are sites in the shared memory. In particular, each thread executes a single coupling operation. Data is partitioned so that each thread process couples of neighbours sites. This is done by arranging the threads in a checkerboard pattern; each thread allocates in its local memory the element corresponding to its position in the checkerboard, keeping it during all the single time step evolution execution. Then, it updates the value of this element and the one in its neighbour in the shared memory, which is determined by the computational step that is currently executed. The single coupling operations are executed in parallel by the threads, but the computational steps are processed in a serialized manner, due to data dependency. Hence, after each computational step there is a synchronization barrier.

4.4 Hybrid Kernel

During the execution of the GPU kernel, the host's CPU is idle while waiting for the GPU to complete the calculations. This CPU time could be used to perform part of the evolution of the state. Furthermore, GPU have less memory than CPU, limiting the size of the quantum system for which the evolution is computed. A hybrid kernel that uses both CPU and GPU address these two issues.

The algorithm calculates the maximum amount of the array sites that can be computed on the GPU. Then an internal area of the array site, with the appropriate size, is sent to the GPU to be calculated. Since CPU and GPU are in charge of evolving different area of the array sites, GPU requires a halo surrounding the internal area while the CPU needs the halo that correspond to the sites at the edge of the internal area. A single time step evolution is performed, using the CPU and GPU kernels described above for the corresponding areas. After that, the internal halos must be updated by performing a communication between CPU and GPU. The GPU receives the halo from the corresponding part of the mesh evolved by the CPU – for the CPU is the other way around.

The kernel also uses a directive-driven parallelism to utilize the power of a multicore system, relying on OpenMP [4]. In the CUDA programming model, each GPU is associated to a single host, which is a single CPU core, hence, in a system with a multicore CPU and a single GPU, only one single CPU core would be used. For this purpose, the CPU kernel is slightly adjusted to use OpenMP, so that the blocks which divide the CPU mesh are processed in parallel using more than one CPU core.

4.5 Distributing the Workload Across a Cluster

The code is designed to run in a distributed multi-node system. The whole mesh is partitioned in blocks called tiles. Each node processes a tile plus its halo. After a single time step evolution the halos between the tiles have to be sent across the nodes. The communication is implemented using MPI [30]. Using a two-dimensional grid of nodes, a tile contains elements of halos belonging to a total of eight other tiles: left, right, top, and bottom neighbours, and also the four diagonal neighbours. To minimize the number of communication requests, a wave pattern is used in the communication: left and right neighbours receive the halo first. This halo has the height of the inner cells of the tile. Then the horizontal halos are sent to the top and bottom neighbours – the width is the full tile width. In this way the appropriate corner elements are propagated to the diagonal neighbours.

The communication is performed asynchronously: each halo is sent at the same time. However, there is a communication barrier between the left-right and top-bottom halo exchange due to data dependency. To achieve better performance, the approach is to overlap communication and evolution as much as possible, evolving the halo first and starting the communication simultaneously to the evolution of the rest of the tile.

The CPU kernels evolve blocks of the tile at its edge, corresponding to the halos. Then the asynchronous left-right halo exchange and the evolution of the inner part of the tile initiates. Once finished the evolution of the inner part, there is the left-right halo exchange barrier. Consequently, the asynchronous top-bottom halo exchange starts and the top-bottom barrier ensure that it finish before swapping the buffers and continuing with the next time step. In this case, the halo exchange cannot be efficiently overlapped with computation, since, after the time step evolution is finished, there is a communication overhead while the vertical halos are exchanged.

As regard the GPU kernel, the communication is performed from the host memory. This increases the complexity, as asynchronous memory copies from the device and to the device have to be performed. To work with such transfer efficiently, the kernel implements *streams*. A GPU stream is basically a queue of tasks for the GPU to perform: kernel execution, and memory copies from and to the device. Tasks in two different streams can overlap with each other, while tasks on the same stream are performed sequentially. When the host queues a task into a GPU stream, it does not have to wait for the task to be completed by the GPU before continuing with the rest of the algorithm. Two streams are implemented in the kernel: queueing the halo computation and the memory copies between host and

device in stream one, and the computation of the inner part of the tile in stream two.

The host starts queueing the evolution of the blocks at the edge of the tile, corresponding with the halo, and the halo copy from device to host in stream one. While the GPU is running the first stream, the host queues the evolution of the rest of the blocks in the second stream. Once the GPU finishes the tasks in the first stream the halo can be exchanged between the nodes, using the wave pattern described above. With this approach, the communication of all halos is efficiently overlapped with the blocks calculation – the task in the second stream. When the halo communication is completed, the host queues the copy of the halos received to the device memory in the first stream. Once the GPU finishes the tasks in each streams, the buffers in the global memories are swapped and the kernel starts over.

The hybrid kernel uses only one stream to queue the tasks for the GPU. The host launch the GPU kernel for the corresponding sites on the internal area of the tile. After that the host – CPU – proceeds to calculate the halo and start the halo exchange. Once the halo exchange is initiated, the sites not in the halo and not on the GPU are evolved by the CPU. Finally, finished the halo exchange, it exchange the internal halo between the part of the tile associated with the GPU and the rest of the matrix.

4.6 Benchmarks

To study the scaling pattern of the execution time as the number of nodes vary, we ran benchmarks on the Minotauro cluster at the Barcelona Supercomputing Center. The Minotauro cluster is comprised of 126 compute nodes, where each node has two Intel Xeon E5649 six-core processors with 12 MB of cache memory, clocked at 2.53 GHz, running a Linux operating system with 24 GB of RAM memory. Every node is equipped with two NVIDIA M2090 graphic cards, each one with 512 CUDA cores and 6 GB of GDDR5 memory. The MPI communication across the nodes is through an Infiniband Network.

The benchmarks ran ten iterations on increasing cluster sizes, using an initial state in a square domain of different lenghts L: 4096, 8192, 16384. The dimensions were chosen so as to fill the device memory on cluster sizes 1, 4 and 16 nodes. GPUs have a better performance when the load is higher, whereas CPUs are less sensitive to the load. For this reason, choosing a matrix dimension that fit the GPUs, we get a fair comparison with respect to the CPU kernels due to their insensitivity to the load. Furthermore, this

Nodes	Matrix size: 4096				Matrix size: 8192			
	CPU	SSE	CUDA	Hybrid	CPU	SSE	CUDA	Hybrid
1	5.30	5.01	0.65	0.84				
2	2.68	2.55	0.38	0.53			1.14	1.56
4	1.37	1.29	0.29	0.34	5.37	5.10	0.66	0.85
8	0.68	0.65	0.20	0.27	2.69	2.57	0.46	0.59
16	0.36	0.34	0.17	0.21	1.35	1.29	0.32	0.40
32	0.20	0.18	0.18	0.22	0.72	0.68	0.27	0.34

Nodes	Matrix size: 16384			
	CPU	SSE	CUDA	Hybrid
4			2.27	3.00
8	1.07		1.24	1.68
16	0.53	5.04	0.74	0.90
32	0.28	2.66	0.49	0.68

Table 4.1: Execution time.

configuration shows how the overall performance decays as the cluster size increases. The times of execution are shown in Table 4.1 and they are also plotted in Figs. 4.11a-4.11c. The results show only the time taken in the main loop of the evolution, as each kernel takes different amounts of time for initialization.

The CPU kernels show an almost linear scaling: as the cluster size is doubled, the execution time is halved. The communication overhead increases with the cluster size, so eventually the advantage of SSE optimization vanishes with large clusters.

The GPU kernel shows a different scaling pattern. When the device memory is loaded to at least 50%, the scaling is close to linear, as in the case of CPU kernels. For lower load, the scaling is less efficient, until the execution time of individual GPUs remains almost constant so the curve flattens out. There is little benefit to gain by this kernel in large clusters with a low resolution of the mesh.

The hybrid kernel shows a pattern similar to the GPU. In this configuration, where the matrix size fit the GPU memory, the execution time is slightly longer. The real advantage is in cases where the device memory is insufficient. In such cases, the speedup can be close to a factor of 2 compared to the CPU kernels.

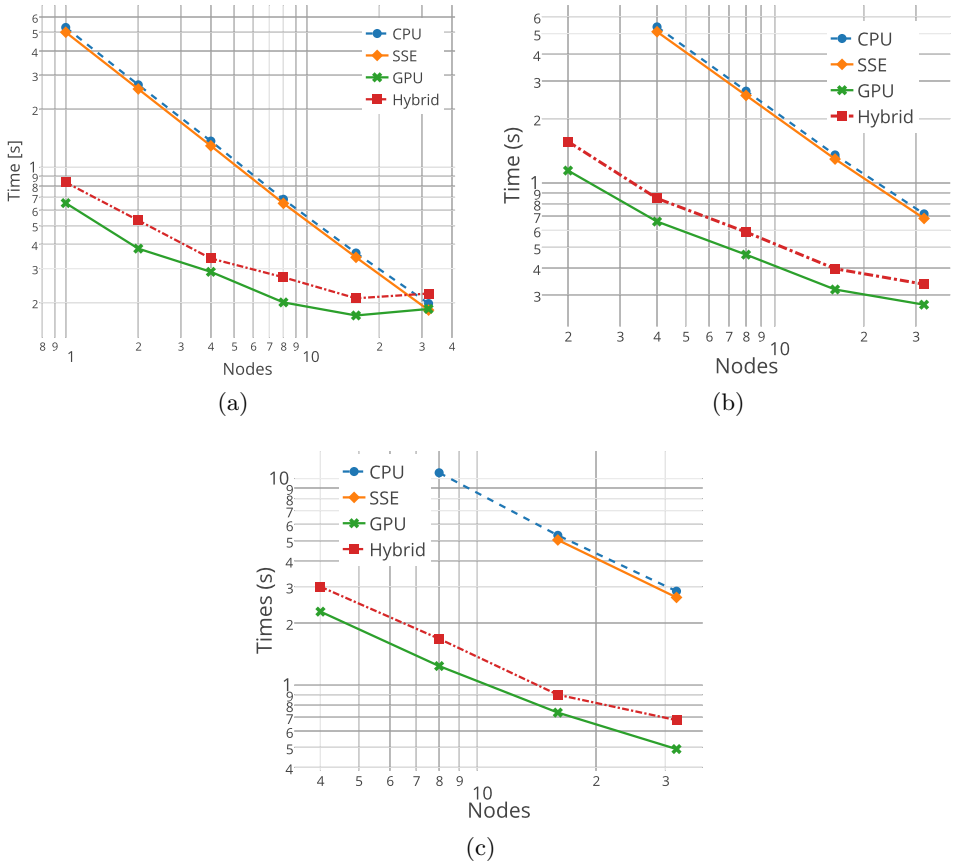


Figure 4.11: Execution time for linear system size: (a) 4096; (b) 8192; (c) 16384.

Chapter 5

Dark Solitons in Bose–Einstein Condensates

As a case study of the code developed, we carried out simulations of the dynamics of a Bose–Einstein condensate (BEC). In particular we reproduced the results of an experimental study performed at the National Institute of Standards and Technology (NIST), in which they investigated the realization and evolution of solitons in a BEC of sodium atoms [7]. In this chapter we briefly introduce the theoretical background of BEC and dark solitons, and proceed with the illustration of the simulation results, comparing them to the results by NIST.

5.1 Theoretical Background

The dynamics of a weakly interacting Bose–Einstein condensate may be described by a nonlinear Schrödinger equation, namely, the Gross–Pitaevskii equation [24, 5, 19]. Suppose we have a N-particle system, comprised of bosons interacting with each other. Using the Hartree approximation, we write the many-body wave function as

$$\psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) = \prod_{i=1}^{N} \phi(\mathbf{r}_i), \tag{5.1}$$

where the single-particle wave function $\phi(\mathbf{r}_i)$ is normalized in the usual way,

$$\int \mathrm{d}\mathbf{r}\, |\phi(\mathbf{r})|^2 = 1. \tag{5.2}$$

For low energy particles, the interaction can be described by an effective potential that in coordinate space corresponds to a contact interaction $U_0\delta(\mathbf{r}-\mathbf{r}')$, where $U_0 = 4\pi\hbar^2 a_s/m$ and a_s is the scattering length of the interaction [24]. Considering an external potential $V(\mathbf{r}_i)$, the Hamiltonian may be written as

$$H = \sum_{i=1}^{N}\left[\frac{\mathbf{p}_i^2}{2m} + V(\mathbf{r}_i)\right] + U_0\sum_{i<j}\delta(\mathbf{r}_i - \mathbf{r}_j). \tag{5.3}$$

Then, the energy of the state is given by

$$E = N\int \mathrm{d}\mathbf{r}\,\left[\frac{\hbar^2}{2m}|\nabla\phi(\mathbf{r})|^2 + V(\mathbf{r})|\phi(\mathbf{r})|^2 + \frac{(N-1)}{2}U_0|\phi(\mathbf{r})|^4\right]. \tag{5.4}$$

It is convenient to introduce the concept of the wave function of the condensed state,

$$\psi(\mathbf{r}) = N^{1/2}\phi(\mathbf{r}), \tag{5.5}$$

so that the normalization condition becomes

$$\int \mathrm{d}\mathbf{r}\,|\psi(\mathbf{r})|^2 = N. \tag{5.6}$$

Moreover, we suppose that $N \gg 1$ so that the energy can be rewritten as,

$$E = \int \mathrm{d}\mathbf{r}\,\varepsilon = \int \mathrm{d}\mathbf{r}\,\left[\frac{\hbar^2}{2m}|\nabla\psi(\mathbf{r})|^2 + V(\mathbf{r})|\psi(\mathbf{r})|^2 + \frac{1}{2}U_0|\psi(\mathbf{r})|^4\right]. \tag{5.7}$$

The equation of motion may be derived from the principle of least action

$$\delta\int_{t1}^{t2}\mathrm{d}t\,L = 0 \tag{5.8}$$

where the Lagrangian L is given by

$$L = \int \mathrm{d}\mathbf{r}\,\left[\frac{\imath\hbar}{2}\left(\psi^*\frac{\partial\psi}{\partial t} - \psi\frac{\partial\psi^*}{\partial t}\right) - \varepsilon\right]. \tag{5.9}$$

Requiring that the variation of the independent variables $\psi(\mathbf{r},t)$ and $\psi^*(\mathbf{r},t)$ vanishes at $t = t_1$ and $t = t_2$, and on any spatial boundaries, from Eq. (5.8) one finds the Gross–Pitaevskii equation:

$$\imath\hbar\frac{\partial\psi(\mathbf{r},t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) + U_0|\psi(\mathbf{r},t)|^2\right]\psi(\mathbf{r},t). \tag{5.10}$$

Formally, Eq. (5.10) is not implemented in our code since the Hamiltonian is not simply comprised of a kinetic term and an external potential term. However, if we denote $\tilde{V}(\mathbf{r}, \psi(\mathbf{r}, t)) = V(\mathbf{r}) + U_0|\psi(\mathbf{r}, t)|^2$, the single time step evolution still has the form (4.2) and the exponential operator $\exp\left(-\frac{\imath}{\hbar}\tilde{V}(\mathbf{r}, \psi(\mathbf{r}, t))\right)$ is still diagonal in the coordinate representation. Thus, we only have to define the potential such that, in the computational step 5 (Fig. 4.6), each site (i, j) is also multiplied by the factor $\exp(-\frac{\imath}{\hbar}U_0|\psi(i, j)|^2)$ where $\psi(i, j)$ is the wave function calculated in the previous computational step.

The ground-state of the system may be found numerically using the imaginary time evolution. However, for sufficiently large number of atoms an accurate expression is obtained using the Thomas–Fermi approximation. Suppose that the contribution of the kinetic energy term is negligible with respect to the one of the interaction terms. Then the ground-state is approximated by the solution of the time independent Gross–Pitaevskii equation deprived of the kinetic term

$$\left[V(\mathbf{r}) + U_0|\psi(\mathbf{r})|^2\right]\psi(\mathbf{r}) = \mu_{TF}\psi(\mathbf{r}), \tag{5.11}$$

where μ_{TF} is the chemical potential. This gives the solution

$$n(\mathbf{r}) = |\psi(\mathbf{r})|^2 = \frac{\mu_{TF} - V(\mathbf{r})}{U_0}, \tag{5.12}$$

for $\mathbf{r}$ such that $V(\mathbf{r}) < \mu_{TF}$. For a BEC trapped in a harmonic potential of the form

$$V(x, y) = \frac{m}{2}(\omega_1^2 x^2 + \omega_2^2 y^2) \tag{5.13}$$

the extension of the condensate wave function in the two directions is given by the two semi-axes

$$R_i = \sqrt{\frac{2\mu_{TF}}{m\omega_i^2}}, \quad i = 1, 2, \tag{5.14}$$

and the particles density has the form of an inverted parabola. Given the normalization condition in Eq. (5.6), the chemical potential is determined as a function of U_0 and the system dimension (see Appendix A).

The Thomas–Fermi approximation gives an accurate description of the bulk properties of the system, but it fails near the edge of the cloud. Provided that $V(\mathbf{r})$ varies slowly, a better solution for the ground state may be found solving the entire time independent Gross–Pitaevskii equation taking the linear term of the external potential at the edge of the cloud [24].

The Gross–Pitaevskii equation has exact analytical solutions in the nonlinear regime. These solutions have the form of solitary waves, called solitons, which are localized disturbances that propagate without changing of form [26, 14, 8, 45]. This phenomenon is due to the balance between dispersion and nonlinearity, which in Eq. (5.10) correspond to the kinetic term and the nonlinear interaction, respectively. Solitons appear in different contexts of science and engineering, such as the dynamics of waves in shallow water [2], transport along DNA and other macromolecules [25], and fiber optic communications [11]. Depending on the details of the governing nonlinear equation, they can be either bright or dark. The former are peaks in the amplitude, while the latter are notches with a characteristic phase step across. In the present work we focus on dark solitons.

For BEC with repulsive interaction between the atoms, solitons present a depression in the density profile – a dark soliton. In a homogeneous BEC, the resulting density profile along the x axis is

$$n(x,t) = n_{min} + (n_0 - n_{min}) \tanh^2 \left(\frac{x - \nu_s t}{\sqrt{2}\xi} \sqrt{1 - \left(\frac{\nu_s}{\nu_0}\right)^2} \right), \qquad (5.15)$$

where n_0 is the unperturbed density, n_{min} is the density at the center of the soliton, ν_s and $\nu_0 = (\frac{nU_0}{m})^{\frac{1}{2}}$ are the speed of the soliton and the speed of sound respectively, and

$$\xi = \frac{\hbar}{(2mn_0U_0)^{1/2}} \qquad (5.16)$$

being the healing length [24]. The speed and the depth of the soliton can be related with each other [26, 14]. Indeed, the soliton speed ν_s can be expressed as

$$\nu_s = \nu_0 \sqrt{\frac{n_{min}}{n}}. \qquad (5.17)$$

Note that the soliton speed is less than the speed of sound; feature that distinguish it from sound waves and excitations. The soliton speed can also be expressed by means of the change in phase of the wave function across it:

$$\nu_s = \nu_0 \cos\left(\frac{\delta}{2}\right). \qquad (5.18)$$

Remarkably, the motion of a soliton in a BEC in an external potential is the same as that of a particle of mass $2m$ in the same potential [24]. Consequently, for a potential having a minimum, the period of the motion of a soliton is $\sqrt{2}$ times that of a particle of mass m in the potential.

5.2 Soliton Simulation

In the experiments carried out in Ref. [7], the generation and propagation of solitons were studied in BEC of sodium atoms, confined in a magnetic trap. The magnetic trap generated a harmonic external potential with frequencies $\omega_x = \sqrt{2}\omega_y = 2\omega_z = 2\pi \cdot 28$ Hz, while the system were composed by $1.7(\pm 0.3) \cdot 10^6$ atoms in the $3S_{1/2}$, $F = 1$, $m_F = -1$ state. In this configuration the scattering length is 2.75 nm [7].

In this experiment, the system is three-dimensional, while our implementation is designed for a two-dimensional system. This would not be a problem if the dynamics of the 3D system was described by three decoupled equations, so that we could independently solve the two corresponding to the variables that span a 2D plane and ignore the other one. The Gross–Pitaevskii equation (5.10) cannot be decoupled due to the non-linear term, so this does not apply to our case. However, our purpose is to study the evolution of a soliton which propagates in one dimension, so we can ignore the dynamic along one of the two dimensions perpendicular to the axis of propagation. An approach to reduce to a two dimensional problem is to make a phenomenological hypothesis about the form of the solution. In the literature [14, 32, 27, 28, 22], one can find several methods to reduce a 3D problem to a 2D one; we adopt the method described in [21]. The condition is that the solution of the reduced system has the same spatial extent of the original one. Since the 3D system is well described by the Thomas–Fermi approximation[1], the extension of the ground state is known (Eq. (5.14)). Then, the approach is to rewrite the coupling constant $U_0^{(2)}$ for the 2D system, such that the chemical potential for the 2D system is the same as the one of the 3D system. This will ensure that the spatial extension of the 2D system will be the same as the 3D system (see Appendix A). For our simulation we reduced the dimension keeping the x and y axes of the 3D system. The resulting Gross–Pitaevskii equation for the 2D system becomes:

$$\imath\hbar\frac{\partial\psi(\mathbf{r},t)}{\partial t} = \left[\sum_{i=1}^{2}\left(-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x_i^2} + \frac{m}{2}\omega_i^2 x_i^2\right) + U_0^{(2)}|\psi(\mathbf{r},t)|^2\right]\psi(\mathbf{r},t), \quad (5.19)$$

where

$$U_0^{(2)} = \frac{\pi}{2}\frac{\mu_{TF}}{N}\left(\bar{R}_{TF}\right)^2, \quad (5.20)$$

[1]The experimental configuration satisfies the conditions to adopt the Thomas–Fermi approximation, namely the BEC is close to the ground state with a large number of atoms, so that kinetic energy term is by far lower than the interaction energy term

with $\bar{R}_{TF} = \sqrt{2\mu_{TF}/m\bar{\omega}^2}$ being the mean Thomas–Fermi radius, $\bar{\omega} = \sqrt{\omega_x\omega_y}$ the geometric mean of the frequencies, and μ_{TF} is the chemical potential in the Thomas–Fermi approximation for the 3D system given by

$$\mu_{TF} = \frac{1}{2}\left(15N\hbar^2\omega_x\omega_y\omega_z\sqrt{m}a_s\right)^{2/5} \tag{5.21}$$

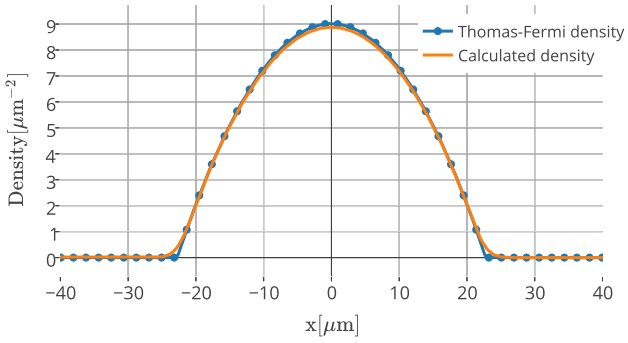
Ground state. According to the experiment, the BEC is initially described by the ground state of Eq. (5.10). They found it to have a spatial extension described by the Thomas–Fermi diameters of $2R_{TF,x} = 45$ μm, $2R_{TF,y} = 64$ μm and $2R_{TF,z} = 90$ μm, with a uniform phase [5]. In the simulation, we approximated the ground state of Eq. (5.19) using imaginary time evolution. We proceeded by taking a Gaussian initial state, evolving it in imaginary time for a fixed number of iterations and then calculating the energy of the resulting state. Repeating this procedure using as initial state the one resulting from the last iteration, the energy decreases converging to the ground state energy. If we call E_i the energy of the state resulting from the iteration i, we stopped the procedure once the $i-$th state satisfied the convergence condition

$$\left|\frac{E_i - E_{i-1}}{E_{i-1}}\right| < 10^{-6}. \tag{5.22}$$

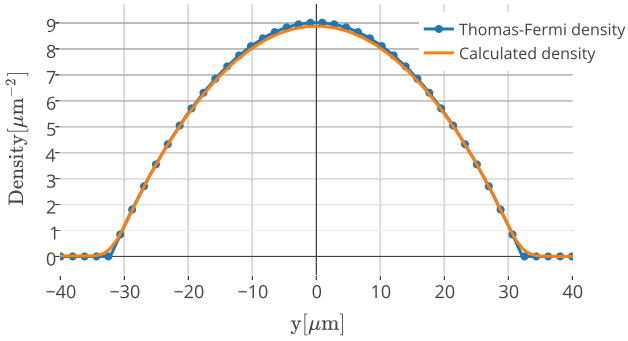
This ensure that the $i-$th state is an accurate approximation of the ground state. In Fig. 5.1a and Fig. 5.1b the profiles of the particle density along the two axes are shown. The particles density of the calculated ground state is in good agreement with the particles density of the Thomas–Fermi approximation and with the experimental results.

Soliton propagation. Solitons can be generated in BEC by phase imprinting. The phase of the ground state is modified by exposing the cloud to pulsed, off-resonant laser light with an intensity pattern $I(x, y)$. The wave function acquires a corresponding phase $\phi(x, y)$ proportional to $I(x, y)$ and the time of exposure T. According to the experiment, they chose T to be short enough so that the atomic motion was negligible (Raman–Nath regime). In this condition, the effect of the pulse can be expressed as a sudden phase imprint, $\psi \to \psi \exp(\imath\phi(x, y))$ [7]. If the center of the BEC correspond to the origin of the axes, the phase imprint performed in [7] can be approximated as

$$\phi(x, y) = \frac{\phi_0}{2}\left[1 + \tanh\left(\frac{x}{l}\right)\right], \tag{5.23}$$



(a)



(b)

Figure 5.1: Calculated ground-state density along the x axis (a) and the y axis (b). The simulation is in good agreement with the Thomas–Fermi approximation. The spatial extension of the calculated ground state corresponds to the experimental results, where $R_{TF,x} = 45$ μm and $2R_{TF,y} = 64$ μm.

where $\phi_0 = 1.5\pi$ and $l = 2$ μm.

According to the experimental configuration, we set our simulation taking as initial state the transformed ground state $\tilde{\psi}_{gs}$, namely

$$\tilde{\psi}_{gs} = \psi_{gs} \exp(\imath\phi(x, y)) \tag{5.24}$$

where ψ_{gs} is the ground state calculated with imaginary time evolution.

The phase imprinting correspond to impressing a momentum to a static ground state, in the region of space where the phase varies. This leads to a collective motion of the system, which corresponds to an oscillation of the

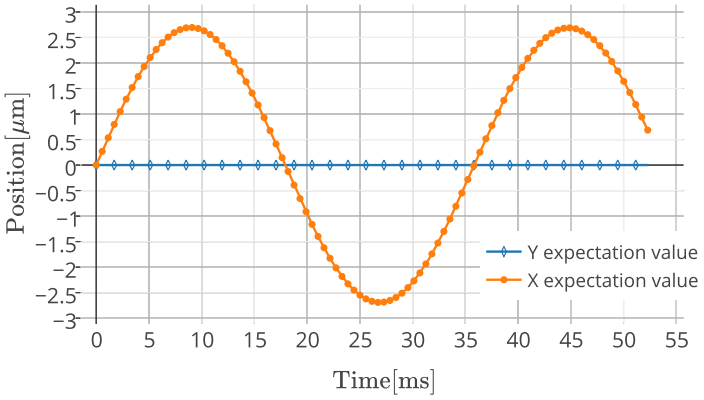


Figure 5.2: Calculated expectation values $\langle X \rangle (t)$ and $\langle Y \rangle (t)$. The calculated oscillation frequency along the x axis, $\omega_x = 2\pi \cdot 27.9$ Hz, is in agrement with the external potential frequency $\omega_x = 2\pi \cdot 28$ Hz. There is no oscillation along the y axis since no impulse is imparted in this direction.

BEC along the x axis. This can be seen in the simulation. Fig. 5.2 illustrates how the expectation value of the position along the x axis varies in time. As expected from the theory, the oscillation along the x axis has the same frequency as the harmonic potential ω_x, while the position along the y axis remains stable. We found a frequency of $\omega_x = 2\pi \cdot 27.9$ Hz, in agreement with the experimental value.

To observe soliton propagation they exploited absorption imaging, measuring the BEC density distribution. Immediately after the phase imprint, they observed a positive density disturbance travelling in the $+x$ direction, and a dark notch left behind it, which travels in the opposite direction – this is the soliton (Fig. 5.5a to 5.5e). The positive disturbance travels with a speed higher than the soliton.

They determined the soliton speed along the x axis, measuring the distance after 5 ms of propagation between the notch and the position of the imprinted phase step. At that time the soliton had not travelled far from the BEC center, so this is a good estimation of the soliton speed at the center of the condensate. They obtained a mean soliton speed of 1.8 ± 0.4 mm/s. This value is lower than the mean speed of sound $\nu_0 = 2.8 \pm 0.1$ mm/s, which ensures that the dark notch is a soliton and not a sound wave [7]. In our simulation, we tracked the position of the soliton along the x axis for 14 ms (Fig. 5.3). We also calculated the mean speed of the notch after it had travelled for 5 ms, obtaining a mean speed of 1.7 mm/s, in agreement

with the experimental value.

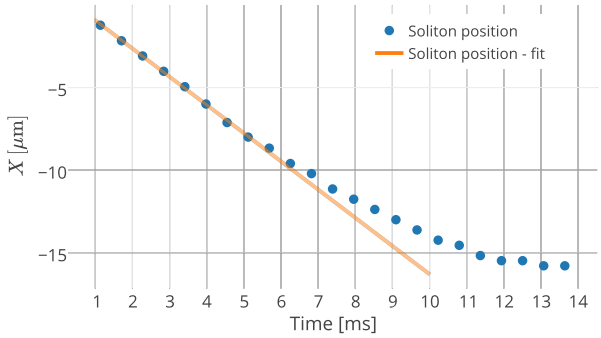


Figure 5.3: Calculated soliton position along the x axis over the time.

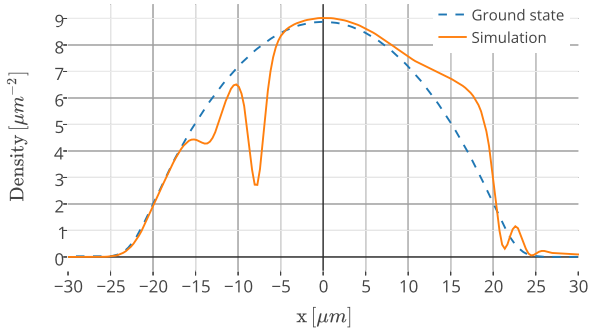
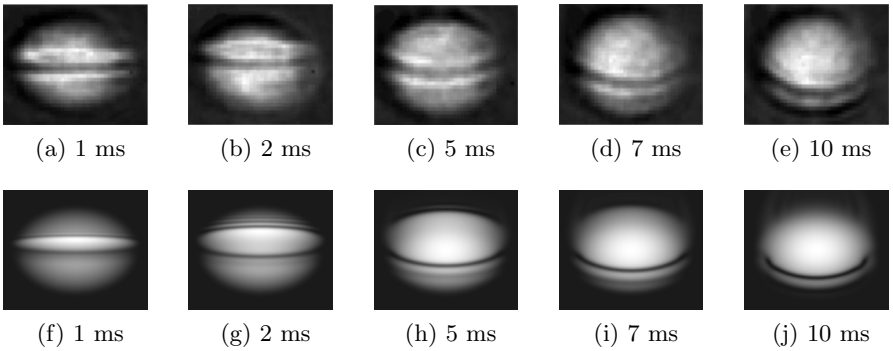


Figure 5.4: Calculated ground state and particles density at $t = 5$ ms along the x axis. The deep soliton is located at $x = -8$ μm, in agreement with the experimental value [7]. Other structures are visible from this figure: a shallow dark soliton at $x = -14$ μm moving to the left; other excitations near $x = 20$ μm moving fast to the right.

The soliton speed is not the same throughout the BEC, as it depends on the particles density (Eq. (5.17)). It is zero at the edge of the BEC, while it reaches its maximum at the center of the BEC. Indeed, rewriting Eq. (5.17) as follows

$$\nu_s = \nu_0 \sqrt{1 - \frac{n - n_{min}}{n}}, \tag{5.25}$$

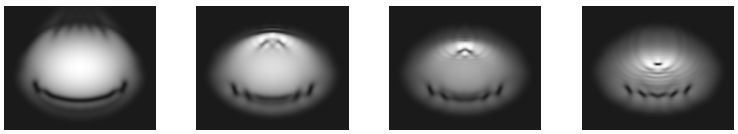
we see that ν_s goes to zero at the edge where both n and $n - n_{min}$ go to zero, whereas the fraction $(n - n_{min})/n$ reaches its lowest value at the center and ν_s is maximum. This implies that the soliton assumes a curved shape, whose curvature increases as the soliton travels far from the center.



(a) 1 ms (b) 2 ms (c) 5 ms (d) 7 ms (e) 10 ms

(f) 1 ms (g) 2 ms (h) 5 ms (i) 7 ms (j) 10 ms

Figure 5.5: Experimental images of the integrated BEC density ((a) to (e)) [7] and calculated density, from our simulation, ((f) to (j)) for various times after the phase imprinting. A positive density disturbance is created and moves rapidly in the $+x$ direction. A dark soliton is left behind moving in the opposite direction at significantly less than the speed of sound.



(a) 14 ms (b) 17 ms (c) 18 ms (d) 20 ms

Figure 5.6: Calculated particles density for various times after the soliton stops. These images show how the soliton breaks up.

This feature of the soliton is observed both in the experiment and in our simulation (Fig. 5.5).

In Fig. 5.4, the particles density profile along the x axis is shown after 5 ms from the phase imprinting. The deep soliton is located at $x = -8$ μm, in agreement with the experimental value [7]. Other structures are visible from this figure: a shallow dark soliton at $x = -14$ μm moving to the left; other excitations near $x = 20$ μm moving fast to the right. These features are not well resolved in the experimental images, but they are in agreement with the simulations in [7].

Since the external potential has a minimum, the soliton is expected to oscillate with a frequency of $\omega_s = \omega_x/\sqrt{2}$. This behaviour was also found by previous simulations [23, 3]. In our case, the soliton should stop after one-quarter of the oscillation time: $T_s/4 = \frac{\pi}{\sqrt{2}\omega_x} = 12.6$ ms, which is in

good agreement with our simulation (Fig. 5.3).

We were able to evaluate a quarter of period of the soliton and not the whole period because after the notch stops it breaks up, becoming rather perturbed (Fig. 5.6). This behaviour is found also in the simulations carried out in [7].

In conclusion, though our simulation is based on the solution of the two-dimensional Gross–Pitaevskii equation, we were able to reproduce the results of both their experiment and simulation, which was based on the solution of the 3D Gross–Pitaevskii equation [7].

Chapter 6

Conclusion

In this Thesis we developed a code for solving the time-dependent Schrödinger equation of a wave function in a two-dimensional space. The code implements a solver for quantum systems described by Hamiltonians with a static external potential and a self-interacting term. Furthermore, the code includes a solver for the imaginary time evolution, to approximate the ground state of the system.

This result has been obtained by extending a recent work of Wittek and Cucchietti [43], in which they developed a solver for a free quantum particle that scales to massively parallel computing clusters. The code implements the algorithm on four different kernels: two cache optimized kernels for central processing unit (CPU) – one is further optimized to use the SSE instructions –, a kernel for general-purpose graphics processing unit (GPU) and a hybrid kernel that use both CPUs and GPUs. The algorithm is based on the second order of the Trotter–Suzuki approximation, which provides an accurate approximation of the evolution operator. Moreover, the approximation leads to an algorithm easy to parallelise that results in an efficient distribution of the workload across the nodes of a cluster. Indeed, the CPU kernels show a linear scaling of the throughput, so that doubling the number of cores, involved in the calculation, results in a halved time of execution. This can lead to very fast simulations on big clusters. On the other hand, GPU and hybrid kernels obtain better performances on a smaller scaled system, which makes them preferable on a single workstations.

We proved the accuracy of our code reproducing the results obtained by an experiment carried out at the National Institute of Standard and Technology (NIST) [7]. We approximated the ground state of a Bose–Einstein condensate (BEC) of sodium atoms, confined in a magnetic trap, and sim-

ulated the evolution of the state that underwent on a phase imprinting. We were able to see the generation of a soliton and other excitations in agreement with the experimental observations.

We achieved this by mean of the approximation described in Ref. [21]. This approximation let us reduce the three dimensional Gross–Pitaevskii equation, which provides a model of the BEC, to a two dimensional model solvable by our code.

The software comes with the General Public License and it can be redistributed under the terms of this license. We developed an application programming interface (API) that exposes the function that performs the evolution. Furthermore, the CPU and SSE kernels are accessible from Python and MATLAB[1].

A clear direction for future extensions of the code is the implementation of the nonlinear Schrödinger equation to all the kernels – at the moment it is only supported on the CPU kernel. These would be useful in many fields of physics, for instance in ultracold atom physics, in which the Gross–Pitaevskii equation plays a major role in describing the systems. In this case, the phenomenology to be studied include vortexes, spin-orbit coupling [9] and quantum thermalization. Other applications may regards the simulations of cloak system and quantum holography.

The method used to rescale a three dimensional problem to a two dimensional one may not be always applicable. This motivates the extension to three dimensions. The variety of possible decomposition strategies in this case is large, and a flexible implementation would be very useful to test out the performance of the different choices.

[1]For more information about how to get the code visit the web site https://github.com/peterwittek/trotter-suzuki-mpi.

Appendix A

Reduction of Dimension: Constant Spatial Extent of the Solution

The understanding of a phenomenon, in science, goes through the solution of a model. Often this task is difficult or even impossible to accomplish without making some hypothesis that simplifies the problem. The approach is to reduce the problem to solve simpler equations with fewer degrees of freedom. A good method is to identify the symmetries of the problem, and rewrite the equations in a new set of variables which makes them decoupled. So that the dynamic can be solved for independent groups of variables or even be ignored for some of them, reducing the problem to a lower number of variables. For instance, the equations for the tri-dimensional dynamic of the two bodies problem can be written as a system of three independent equations, using the appropriate variables.

For many models this approach is not suitable, as in the case of the Gross–Pitaevskii equation for a self-interacting BEC. Another procedure consist of making a posteriori (phenomenological) hypotheses, that let us deal with a simpler problem with fewer variables. In this work we reduce a 3D Gross–Pitaevskii equation to a 2D equation, requiring that the solution of the 2D equation has the same spatial extent [21].

The general d-dimensional Gross–Pitaevskii equation, with an anisotropic harmonic potential, is written as:

$$\imath\hbar\frac{\partial\psi(\mathbf{r},t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\sum_{i=1}^{d}\frac{\partial^2}{\partial x_i^2} + \frac{m}{2}\sum_{i=1}^{d}\omega_i^2 x_i^2 + U_0^{(d)}|\psi(\mathbf{r},t)|^2\right]\psi(\mathbf{r},t), \quad \text{(A.1)}$$

where the form of $U_0^{(d)}$ depends on the dimension of the problem; for $d = 3$ we have:

$$U_0^{(3)} = \frac{4\pi\hbar^2 a_s}{m}. \tag{A.2}$$

In the Thomas–Fermi approximation, the wave function of a stationary state satisfies the condition

$$|\psi_{TF}(\mathbf{r})|^2 = \frac{\mu_{TF} - V(\mathbf{r})}{U_0^{(d)}} \text{ if } \mu_{TF} > V(\mathbf{r}), \tag{A.3}$$

so, given the external potential, the spatial extent of the system is determined by the chemical potential μ_{TF}. From Eq. (A.3) we see that, in the case of an anisotropic harmonic potential, the spatial extent of a reduced system will be the same as the complete one if the chemical potential is the same.

Furthermore, in the Thomas–Fermi approximation it is possible to express the coefficient of the self-interaction, U_0, as a function of the chemical potential, due to the normalization condition of the wave function, namely:

$$\begin{aligned}
N &= \int_V \mathrm{d}^d\mathbf{r}\, |\psi_{TF}(\mathbf{r})|^2 \\
&= \int_V \mathrm{d}^d\mathbf{r}\, \frac{1}{U_0^{(d)}} \left[\mu_{TF} - \frac{m}{2}\sum_{i=1}^{d} \omega_i^2 x_i^2\right] \\
&= \frac{\mu_{TF}}{U_0^{(d)}} \left(\bar{R}_{TF}\right)^d \int_{S_1^{(d)}} \mathrm{d}^d\mathbf{r}(1 - \mathbf{r}^2) \\
&= \frac{\mu_{TF}}{U_0^{(d)}} \left(\bar{R}_{TF}\right)^d \frac{2\pi^{d/2}}{\Gamma(d/2)} \int_0^1 \mathrm{d}r\, r^{d-1}[1 - r^2] \\
&= \frac{\mu_{TF}}{U_0^{(d)}} \left(\bar{R}_{TF}\right)^d \frac{2\pi^{d/2}}{\Gamma(d/2)} \left(\frac{1}{d} - \frac{1}{d+2}\right) \\
&= \frac{\mu_{TF}}{U_0^{(d)}} \left(\bar{R}_{TF}\right)^d \frac{\pi^{d/2}}{\Gamma(d/2+2)},
\end{aligned} \tag{A.4}$$

where N is the number of particles, $\bar{R}_{TF} = \sqrt{2\mu_{TF}/m\bar{\omega}^2}$ is the mean Thomas–Fermi radius, $\bar{\omega} = \sqrt[d]{\prod_{i=1}^{d} \omega_i}$ is the geometric mean of the frequencies and $S_1^{(d)}$ is the d-dimensional sphere. This gives us the formula

$$U_0^{(d)} = \frac{\mu_{TF}}{N} \left(\bar{R}_{TF}\right)^d \frac{\pi^{d/2}}{\Gamma(d/2+2)}, \tag{A.5}$$

that for $d = 2$ gives

$$U_0^{(2)} = \frac{\pi}{2} \frac{\mu_{TF}}{N} \left(\bar{R}_{TF}\right)^2 . \tag{A.6}$$

Combining Eq. (A.6) and the constant of the chemical potential we can reduce the problem from three to two dimension. The resulting equation for the reduced system will be

$$\imath\hbar \frac{\partial \psi(\mathbf{r}, t)}{\partial t} = \left[-\frac{\hbar^2}{2m} \sum_{i=1}^{2} \frac{\partial^2}{\partial x_i^2} + \frac{m}{2} \sum_{i=1}^{2} \omega_i^2 x_i^2 + \frac{\pi}{2} \frac{\mu_{TF}}{N} \left(\bar{R}_{TF}\right)^2 |\psi(\mathbf{r}, t)|^2 \right] \psi(\mathbf{r}, t).$$

Bibliography

[1] C. S. Bederián and A. D. Dente. Boosting quantum evolutions using Trotter–Suzuki algorithms on GPUs. In *Proceedings of HPCLatAm-11, 4th High-Performance Computing Symposium*, Cordoba, Argentina, December 2011.

[2] R.K. Bullough. "The Wave" "Par Excellence", the solitary progressive great wave of equilibrium of the fluid: An early history of the solitary wave. In Muthusamy Lakshmanan, editor, *Solitons*, Springer Series in Nonlinear Dynamics, pages 7–42. Springer Berlin Heidelberg, 1988.

[3] T. Busch and J. R. Anglin. Mossbauer effect for dark solitons in Bose–Einstein condensates. *arXiv:cond-mat/9809408v1*, 1998.

[4] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[5] F. Dalfovo, S. Giorgini, L. P. Pitaevskii, and S. Stringari. Theory of Bose–Einstein condensation in trapped gases. *Reviews of Modern Physics*, 71:463–512, 1999.

[6] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26:149–157, 1999.

[7] J. Denschlag, J. E. Simsarian, D. L. Feder, C. W. Clark, L. A. Collins, J. Cubizolles, L. Deng, E. W. Hagley, K. Helmerson, W. P. Reinhardt, S. L. Rolston, B. I. Schneider, and W. D. Phillips. Generating solitons by phase engineering of a Bose–Einstein condensate. *Science*, 287(5450):97–101, 2000.

[8] R. Dum, J. I. Cirac, M. Lewenstein, and P. Zoller. Creation of dark solitons and vortices in Bose–Einstein condensates. *Physical Review Letters*, 80:2972–2975, 1998.

[9] V. Galitski and B. Spielman. Spin–orbit coupling in quantum gases. *Nature*, 494:49–54, 2013.

[10] J. Handy. *The cache Memory Book*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Academic Press, Massachusetts, 2nd edition, February 1998.

[11] A. Hasegawa. *Optical Solitons in Fibers*. Springer–Verlag, Berlin, 2nd edition, 1990.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Inc., San Francisco, CA, USA, 2nd edition, 1996.

[13] K. Huang. *Statistical Mechanics*. John Wiley and Sons, 2nd edition, 1928.

[14] A. D. Jackson, G. M. Kavoulakis, and C. J. Pethick. Solitary waves in clouds of Bose–Einstein condensed atoms. *Physical Review A*, 58:2417–2422, 1998.

[15] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60. ACM, 2006.

[16] T. Kato. Trotter's product formula for an arbitrary pair of self-adjoint contraction semigroups. In Academic Press, editor, *Topics in functional analysis (essays dedicated to M. G. Krein on the occasion of his 70th birthday)*, volume 3 of *Advances in mathematics: supplementary studies*, pages 185–195. New York, 1978.

[17] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[18] L. D. Landau and E. M. Lifshitz. *Mechanics*, volume 1. USSR Academy of Sciences, Moscow, 3rd edition, 1981.

[19] M. Lewenstein, A. Sanpera, and V. Ahufinger. *Ultracold Atoms in Optical Lattices: Simulating Quantum Many-Body Systems*, volume 54. Oxford University Press, Oxfordshire, 2nd edition, 2012.

[20] B. Madden. Using CPPUnit to implement unit testing. In M. Dickheiser, editor, *Game Programming Gems*, volume 6 of *Game Development Series*, chapter 1.7. Charles River Media, Massachusetts, 1st edition, April 2006.

[21] P. Massignan. Modello unidimensionale per lo studio dell'espansione di condensati di Bose–Einstein da reticoli ottici. Master's thesis, Università Degli Studi Di Milano, 2001.

[22] P. Massignan and M. Modugno. One-dimensional model for the dynamics and expansion of elongated Bose–Einstein condensates. *Physical Review A*, 67:023614, 2003.

[23] A. E. Muryshev, H. B. van Linden van den Heuvell, and G. V. Shlyapnikov. Stability of standing matter waves in a trap. *Physical Review A*, 60:R2665–R2668, 1999.

[24] C. J. Pethick and H. Smith. *Bose–Einstein condensation in dilute gases*. Cambridge University Press, Cambridge, 2002.

[25] M. Peyrard. *Molecular Excitations in Biomolecules*. Springer–Verlag, New York, 1995.

[26] W. P. Reinhardt and C. W. Clark. Soliton dynamics in the collisions of Bose–Einstein condensates: an analogue of the Josephson effect. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 30(22):L785, 1997.

[27] L. Salasnich. Pulsed quantum tunneling with matter waves. *Laser Physics*, 12:198–202, 2002.

[28] L. Salasnich, A. Parola, and L. Reatto. Effective wave equations for the dynamics of cigar-shaped and disk-shaped Bose condensates. *Physical Review A*, 65:043614, 2002.

[29] M. F. Smith. *Software Prototyping: Adoption, Practice and Management*. McGraw–Hill, New York, 1991.

[30] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI-The Complete Reference: The MPI Core*. MIT Press Cambridge, Massachusetts, 1998.

[31] L. Sophus and E. Friedrich. Theorie der transformationsgruppen. *Leipzig*, 1888, 1890, 1893.

[32] M. J. Steel and W. Zhang. Bloch function description of a Bose–Einstein condensate in a finite optical lattice. *arXiv preprint cond-mat/9810284*, 1998.

[33] M. Suzuki. Decomposition formulas of exponential operators and Lie exponentials with some applications to quantum mechanics and statistical physics. *Journal of Mathematical Physics*, 26(4):601–612, 1985.

[34] M. Suzuki. Fractal decomposition of exponential operators with applications to many-body theories and Monte Carlo simulations. *Physics Letters A*, 146(6):319–323, 1990.

[35] M. Suzuki. General theory of higher-order decomposition of exponential operators and symplectic integrators. *Physics Letters A*, 165(5–6):387–395, 1992.

[36] M. Suzuki. Quantum Monte Carlo methods and general decomposition theory of exponential operators and symplectic integrators. *Physica A: Statistical Mechanics and its Applications*, 205(1):65–79, 1994.

[37] M. Suzuki. Hybrid exponential product formulas for unbounded operators with possible applications to Monte Carlo simulations. *Physics Letters A*, 201(5–6):425–428, 1995.

[38] M. Suzuki. Algebraic formulation of quantum analysis and its applications to Laurent series operator functions. *Physics Letters A*, 224(6):337–345, 1997.

[39] M. Suzuki. Mathematical basis of computational statistical physics and quantum analysis. *Computer Physics Communications*, 127(1):32–36, 2000.

[40] M. Suzuki and K. Umeno. Higher-order decomposition theory of exponential operators and its applications to QMC and nonlinear dynamics. In D. P. Landau, K. K. Mon, and H. B. Schuttler, editors, *Computer Simulation Studies in Condensed-Matter Physics*, volume 6. Springer.

[41] Trotter and Hale F. On the product of semi-groups of operators. *Proceedings of the American Mathematical Society*, 10(4):545–551, 1959.

[42] K. Umeno and M. Suzuki. Symplectic and intermittent behaviour of Hamiltonian flow. *Physics Letters A*, 181(5):387–392, 1993.

[43] P. Wittek and F. M. Cucchietti. A second-order distributed Trotter–Suzuki solver with a hybrid CPU-GPU kernel. *Computer Physics Communications*, 184(4):1165–1171, 2013.

[44] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] O. Zobay, S. Pötting, P. Meystre, and E. M. Wright. Creation of gap solitons in Bose–Einstein condensates. *Physical Review A*, 59:643–648, 1999.

Extended Computational Kernels in a Massively Parallel Implementation of the Trotter–Suzuki Approximation

Peter Wittek[a,b], Luca Calderaro[c]

[a] *ICFO-The Institute of Photonic Sciences, Av. Carl Friedrich Gauss, 3, 08860 Castelldefels (Barcelona), Spain*
[b] *University of Borås, Allegatan 1, 50190 Borås, Sweden*
[c] *University of Padua, Riviera T. Livio 6, 35123 Padua, Italy*

Abstract

We extended a parallel and distributed implementation of the Trotter–Suzuki algorithm for simulating quantum systems to study a wider range of physical problems and to make the library easier to use. The new release allows periodic boundary conditions, many-body simulations of non-interacting particles, arbitrary stationary potential functions, and imaginary time evolution to approximate the ground state energy. The new release is more resilient to the computational environment: a wider range of compiler chains and more platforms are supported. To ease development, we provide a more extensive command-line interface, an application programming interface, and wrappers from high-level languages.

Keywords: Quantum Evolution, Trotter–Suzuki Algorithm, Parallel and Distributed Computing, GPU Computing

NEW VERSION PROGRAM SUMMARY

Program Title: Trotter–Suzuki-MPI
Catalogue identifier: AEXL_v1_0
Licensing provisions: GNU General Public License, version 3
Programming language: C++, CUDA, Python, MATLAB
Computer: x86-64
Operating system: Linux
RAM: 5 MByte–512 GBytes
Number of processors used: 1–64 in a single node, more in a cluster
Supplementary material:
Keywords: GPU Computing, MPI, Quantum Evolution, Trotter–Suzuki Algorithm, Hybrid Kernel

Classification: 4.12 Computational Methods: Other Numerical Methods
External routines/libraries: OpenMP, MPI, CUDA
Subprograms used:
Journal reference of previous version: Computer Physics Communications 184 (4), 1165–1171 (2013)
Does the new version supersede the previous version?: Yes. The original version is not held in the CPC Program Library but can be obtained from `https://github.com/peterwittek/trotter-suzuki-mpi`.
Nature of problem:
The evolution of a general quantum system is described by the time-dependent Schrödinger equation. The solution of this equation involves calculating a matrix exponential, which is formally simple, but computer implementations must consider several factors to achieve both high performance and high accuracy.
Solution method:
The Trotter–Suzuki approximation leads to an efficient algorithm for solving the time-dependent Schrödinger equation [1, 2]. The implementation relies on high-performance parallel kernels in a distributed environment to maximize the computational power of this algorithm [3, 4].
Reasons for the new version:
The computational kernels were generalized to be able to address a much wider range of physics problems. Furthermore, the code has been modularized to make development easier, providing both a command-line and an application programming interface. High-level wrappers from Python and MATLAB provide further ease of use.
Summary of revisions:

1. The implementation was generalized to include a richer variety of physics problems. The problem can have periodic boundary conditions. Many-body simulations of non-interacting particles became possible extension. We can define an arbitrary stationary potential function. The convenience function `expect_values` helps to obtain expectation values.
2. Imaginary time evolution was implemented to find the ground state before starting the simulation. To avoid imposing the overhead of conditional branching in the most computationally intense parts of the code, some of the core kernel functions were duplicated to include the imaginary time evolution.
3. Most of the functionality is exposed through a command-line interface (CLI) for convenience. This allows specifying the files of the initial state and the potential, the parameters of the Hamiltonian, and further parameters related to the simulation, such as the computational kernel to use and the frequency at which snapshots should be written to the disk.

	CPU	SSE	GPU	Hybrid
old release	6512259	4835126	2566198	1758244
new release	7721790	6634026	3110414	2120868
ratio of slow down	19%	37%	21%	20%

Table 1: Comparison of execution time between the old and the new release. The unit is microseconds.

4. The full functionality of the implementation is exposed as an application programming interface (API) through the 'trotter' function. This allows for integrating the simulation in a larger MPI programme and it is also useful for initializing the state and the potential without having files on the disk. To demonstrate the use of the API, several examples are provided with the code.
5. To further ease development, we redesigned the structure of the implementation, making it more modular. We also introduced a unit testing framework to avoid regression.
6. We improved the testing of MPI dependencies by the configure script and allow compilation without MPI. We also improved the treatment of Intel and Visual C++ compilers.
7. We developed wrappers for Python and MATLAB for the CPU kernel for a high-level interface with the library.

Restrictions:
The vectorized CPU kernel must have a tile width that is divisible by two. This puts a constraint on the possible matrix sizes for this kernel. For instance, running twelve MPI threads in a 4×3 configuration, the dimensions must be divisible by six and eight.
Unusual features:
The library currently only supports the CPU kernel under Windows. The Python and MATLAB wrappers support the CPU and SSE kernels.
Additional comments:
The high-performance kernels were independently extended to study spin dynamics [5]. It remains for future work to include lattice models in this implementation.
Running time:
The generalization slightly altered the memory access patterns of the computational kernels, yielding performance penalty of approximately 20 % compared to the previous version (Table 1). The scaling properties did not change and we see a near-optimal scaling when increasing the number of nodes. The actual running time depends on the system size and the duration to be simulated, and the com-

putational resources. It can range from a few seconds to several days.

Acknowledgment: LC was sponsored by the Erasmus+ programme. Further calculations were sponsored by the Spanish Supercomputing Network (FI-2015-2-0023).

[1] H. Trotter, On the product of semi-groups of operators, Proceedings of the American Mathematical Society 10 (1959) 545–551.

[2] M. Suzuki, Decomposition formulas of exponential operators and Lie exponentials with some applications to quantum mechanics and statistical physics, Journal of Mathematical Physics 26 (1985) 601.

[3] C. Bederián, A. Dente, Boosting quantum evolutions using Trotter–Suzuki algorithms on GPUs, in: Proceedings of HPCLatAm-11, 4th High-Performance Computing Symposium, Córdoba, Argentina, 2011.

[4] P. Wittek, F. Cucchietti, A second-order distributed Trotter–Suzuki solver with a hybrid CPU-GPU kernel, Computer Physics Communications 184 (4) (2013) 1165–1171. doi:10.1016/j.cpc.2012.12.008.

[5] A. D. Dente, C. S. Bederián, P. R. Zangara, H. M. Pastawski, GPU accelerated Trotter–Suzuki solver for quantum spin dynamics, arXiv:1305.0036.